MICROSOFT®

GW-BASIC

Microsoft®
GW-BASIC
Interpreter

for the MS-DOS Operating System

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 410130001-320-O00-0286

Contents

Contents

6 Language Reference 53

Contents

Tables

Figures

Chapter 1

Introduction

In 1975, Microsoft wrote the first BASIC interpreter for microcomputers. Today, Microsoft BASIC has well over one million installations and is used in many operating environments. It's the BASIC you will find on all of the most popular microcomputers. Many users, manufacturers, and software vendors have written application programs in Microsoft™ BASIC.

The BASIC interpreter is a general-purpose programming language: it is effective for many applications, including business, science, games, and education. It is interactive; that is, without writing a program, a user can perform processes, calculations, and program testing.

Microsoft GW™ -BASIC is the most extensive implementation of Microsoft BASIC available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC, and supports many features rarely found in other BASIC interpreters. In addition, the Microsoft GW-BASIC Interpreter has sophisticated screen handling, graphics, and structured programming features that are especially suited for application development.

1.1 Notational Conventions

The following syntax notation is used in this manual:

italics	Italics indicate places where specific terms, supplied by the user, will appear. For example, in
	BASIC *filename*
	filename is italicized to indicate that this is a general form, and that a specific filename will be substituted for *filename* in an actual command.
	Italics are also used occasionally in the text for emphasis.
`screen text`	This special typeface is used for example programs, screen output, and command lines, for example:
	`10 PRINT "This is a sample program line."`
`input text`	This special typeface is used to indicate input you enter in response to a prompt. In the following example, "sample" is entered in response to the "Source filename" prompt:
	`Source filename [.BAS]: sample`

CAPITALS
    Capital letters are used in the text to indicate Microsoft BASIC keywords. This is a convention for the documentation only; you don't have to enter these words in capital letters.

[[*item*]]
    Square brackets indicate that the enclosed item is optional. For example, in

    .CALL *name* [[*argumentlist*]]

    the brackets around *argumentlist* indicate you are not required to supply this when you use the CALL statement.

{ *item1*|*item2*|*item3*... }
    Braces indicate that you have a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

|
    A vertical bar separates the choices within braces. At least one of the entries separated by bar(s) must be chosen unless the entries are also enclosed in square brackets.

...
    Ellipses indicate that an entry may be repeated as many times as needed or desired.

    Vertical ellipses are also used in program examples to indicate that a portion of the program is omitted. For instance, two statements are shown in the following example. The ellipses between the statements indicate that intervening program lines occur but are not shown:

GOSUB
.
.
RETURN

1.2   Resources For Learning BASIC

This manual provides complete instructions for using Microsoft BASIC. However, no training material for BASIC programming has been provided. If you are new to BASIC or need help in learning programming, we suggest you read one of the following:

Dwyer, Thomas A. and Critchfield, Margot. *BASIC and the Personal Computer.* Reading, Mass.: Addison-Wesley Publishing Co., 1978.

Albrecht, Robert L., Finkel, LeRoy, and Brown, Jerry. *BASIC.* New York: Wiley Interscience, 2nd ed., 1978.

Billings, Karen and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: Dilithium Press, 1979.

Coan, James. *Basic BASIC.* Rochelle Park, N.J.: Hayden Book Company, 1978.

Chapter 2
Using the BASIC Interpreter
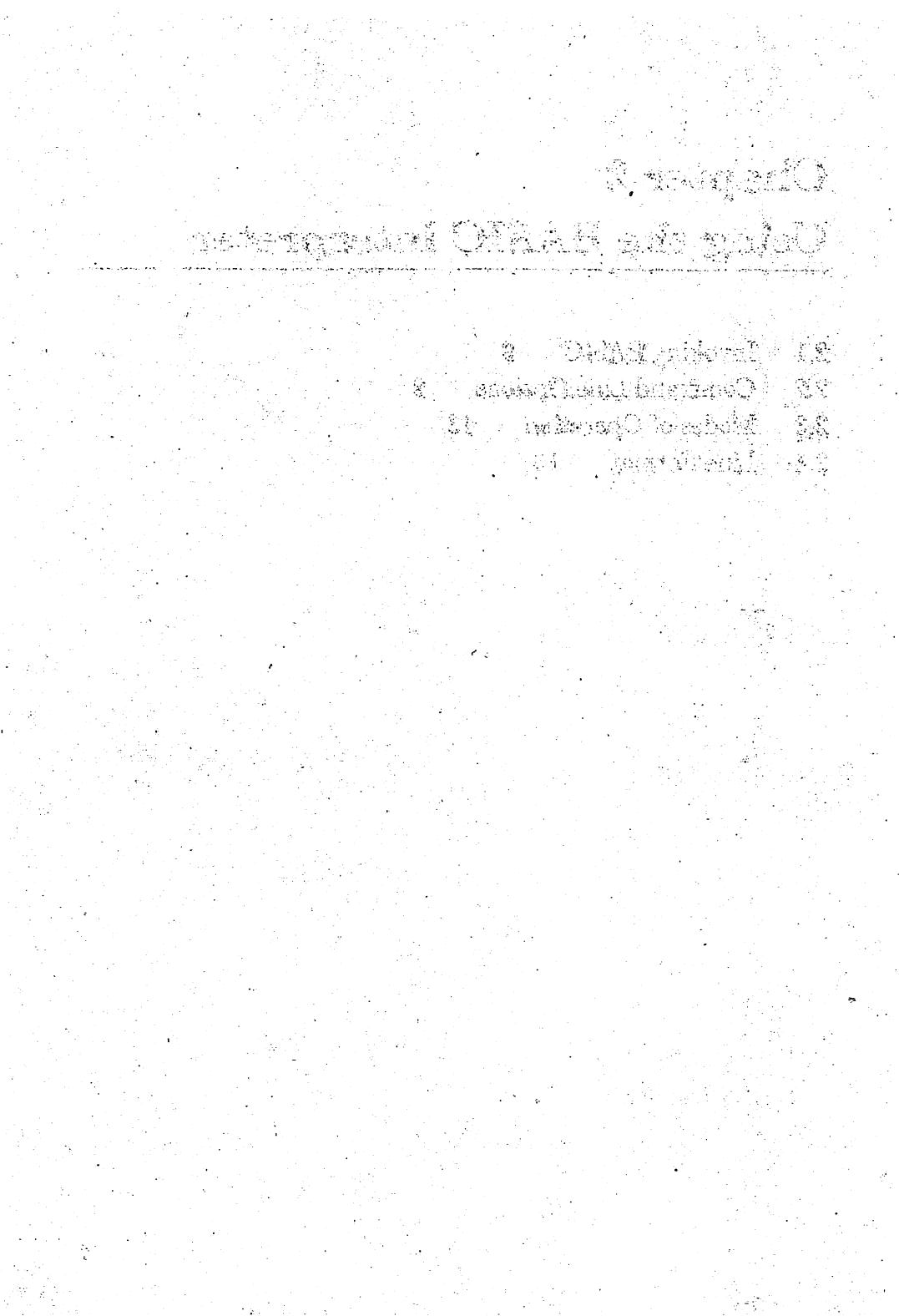
2.1 Invoking BASIC

To begin operating the BASIC interpreter, boot DOS and then enter:

BASIC

To begin operating a specific program as soon as BASIC has started, enter:

BASIC *filespec*

where *filespec* is a file name preceded by an optional device name, and followed by an optional extension.

For example, to start the program FILE.BAS which is on disk drive A, enter:

BASIC "A:FILE.BAS"

2.2 Command Line Options

The BASIC operating environment may be altered by specifying options on the command line. The format of BASIC's command line is:

BASIC *options* ⟦*<stdin*⟧ ⟦*>stdout*⟧ ⟦*filespec*⟧

<stdin   BASIC input is redirected from the file specified by stdin. When present, this syntax must appear before any options.

>stdout   BASIC output is redirected to the file specified by stdout. When present, this syntax must appear before any options. If two greater-than signs appear (">>"), the output is appended to an existing output file. If an existing file is to be written to, this is the way to prevent that file from being overwritten.

filespec   This is the file specification of a BASIC program. If *filespec* is present, BASIC proceeds as if a

RUN *filespec*

command were given after initialization is complete. This allows BASIC programs to be initiated by a batch file by putting this form of the command line in an

9

AUTOEXEC.BAT file. Programs run in this manner will need to exit via the **SYSTEM** statement in order to allow the next command from the **AUTOEXEC.BAT** file to be executed.

The set of legal options follows:

/F:numberoffiles

This option is ignored unless the **/I** option is specified on the command line. Please see **/I** below.

If this option and the **/I** option are present, the maximum number of files that may be open simultaneously during the execution of a BASIC program is set to *numberoffiles*. Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. The data buffer size may be altered with the **/S:** option. If the **/F** option is omitted, the number of files is set to 3.

The number of open files that DOS supports depends upon the value of the **FILES** parameter in the **CONFIG.SYS** file. Ten is the recommended number for BASIC. Keep in mind that the first 3 are taken by Stdin, Stdout, Stderr, Stdaux, and Stdprn. One additional handle is needed by BASIC for **LOAD, SAVE, CHAIN, NAME,** and **MERGE.** This leaves six for BASIC file I/O, thus **/F:6** is the maximum supported by DOS when

```
FILES=10
```

appears in the CONFIG.SYS file.

Attempting to **OPEN** a file after all the file handles have been exhausted will result in a "Too many files" error.

/S:lrecl

This option is ignored unless the **/I** option is specified on the command line. Please see **/I** below.

If this option and the **/I** option are present, then the maximum record size allowed for use with random files is set to *lrecl*.

Note

The record size option to the **OPEN** statement cannot exceed this value.

If the /S: option is omitted, the record size defaults to 128 bytes.

/C:*buffersize*

If present, this option controls RS-232 Communications. If RS-232 cards are present, /C:0 disables RS232 support. Any subsequent I/O attempts will result in a "Device Una-vailable" error. Specifying /C:*n* allocates space for com-munications buffers. If present, this option causes the double-precision transcendental math package to remain resident. If omitted, this package is discarded and the space is freed for program use.

/I

BASIC is able to dynamically allocate space required to support file operations. For this reason, BASIC does not need to support the /S and /F options. However, certain applications have been written in such a manner that cer-tain BASIC internal data structures must be static. To provide compatibility with these BASIC programs, BASIC statically allocate space required for file operations based on the /S and /F options when the /I option is specified.

/M:[[*highmemlocation*]] [[,*maxblocksize*]]

When present, this option sets the highest memory loca-tion that will be used by BASIC. BASIC will attempt to allocate 64K of memory for the data and stack segment. If machine language subroutines are to be used with BASIC programs, use the /M: option to set the highest location that BASIC can use. When omitted or equal to zero, BASIC attempts to allocate all it can up to a maximum of 65536 bytes.

If you intend to load routines or data above the highest lo-cation that BASIC can use, then use the optional parame-ter *maxblocksize* to preserve space. This is necessary if you intend to use the **SHELL** statement. Failure to do so will result in **COMMAND.COM** being loaded on top of your routines when a **SHELL** statement is executed.

The *maxblocksize* argument must be in paragraphs (i.e. byte multiples of 16). When omitted, &H1000 (4096) is as-sumed. This allocates 65536 bytes (65536= 4096 x 16) for BASIC's data and stack segment. For example, if you wanted 65536 bytes for BASIC and 512 bytes for machine language subroutines, then use /M: , &H1010 (4096 para-graphs for BASIC plus 16 paragraphs for your routines).

This option can also be used to shrink the BASIC block in order to free more memory for shelling other programs.

/M:,2048 allocates 32768 bytes for data and stack.
/M:32000,2048 allocates 32768 bytes maximum, but
BASIC will only use the lower 32000. This leaves 768 bytes
for the user.

For the above option arguments, *numberoffiles, lrecl, buffersize, highmemlo-cation,* and *maxblocksize* are numbers that may be decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

Examples

A>BASIC PAYROLL

The above example says to use 64K of memory and three files, and to load and execute PAYROLL.BAS.

A>BASIC INVENT/I/F:6

The above example says to use 64K of memory and six files, and then load and execute INVENT.BAS.

A>BASIC /C:0/M:32768

The above example says to disable RS-232 support and use only the first 32K of memory. The memory above that is free for use by other routines.

A>BASIC /I/F:4/S:512

The above example says to use four files and allow a maximum record length of 512 bytes.

A>BASIC TTY/C:512

The above example says to use 64k of memory and 3 files. Also to allocate 512 bytes to RS-232 receive buffers and 128 bytes to transmit buffers, and to load and execute TTY.BAS.

2.3 Modes of Operation

The Microsoft BASIC Interpreter may be used in either of two modes: direct mode or indirect mode.

In direct mode, statements and commands are executed as you enter them. Commands are not preceded by line numbers. After each direct statement followed by a carriage return, the screen displays the Ok prompt. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. Direct mode is useful for debugging and for using the BASIC Interpreter as a calculator for quick computations that do not require a complete program.

Indirect mode is used for entering programs. Program lines are preceded by line numbers and may later be stored in memory. The program stored in memory is executed by entering the RUN command.

2.4 Line Format

Microsoft BASIC program lines have the following format:

nnnnn statement[:statement...]return

More than one BASIC statement may be placed on a line, but each must be separated from the last by a colon.

A Microsoft BASIC program line always begins with a line number and ends with a RETURN. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in the program itself and when editing to refer to lines to be edited. Line numbers must be in the range 0 to 65529.

A line may contain a maximum of 255 characters.

With the interpreter, you can extend a logical line over more than one physical line by entering a *linefeed*. The *linefeed* lets you continue typing a logical line on the next physical line without entering a RETURN. Alternatively, you may type up to 255 characters on a logical line without issuing either a line feed or a carriage return; the text is wrapped and continues on the next physical line.

A period (.) may be used in **EDIT, LIST, AUTO,** and **DELETE** commands to refer to the current line.

Chapter 3
Writing Programs Using the BASIC Editor
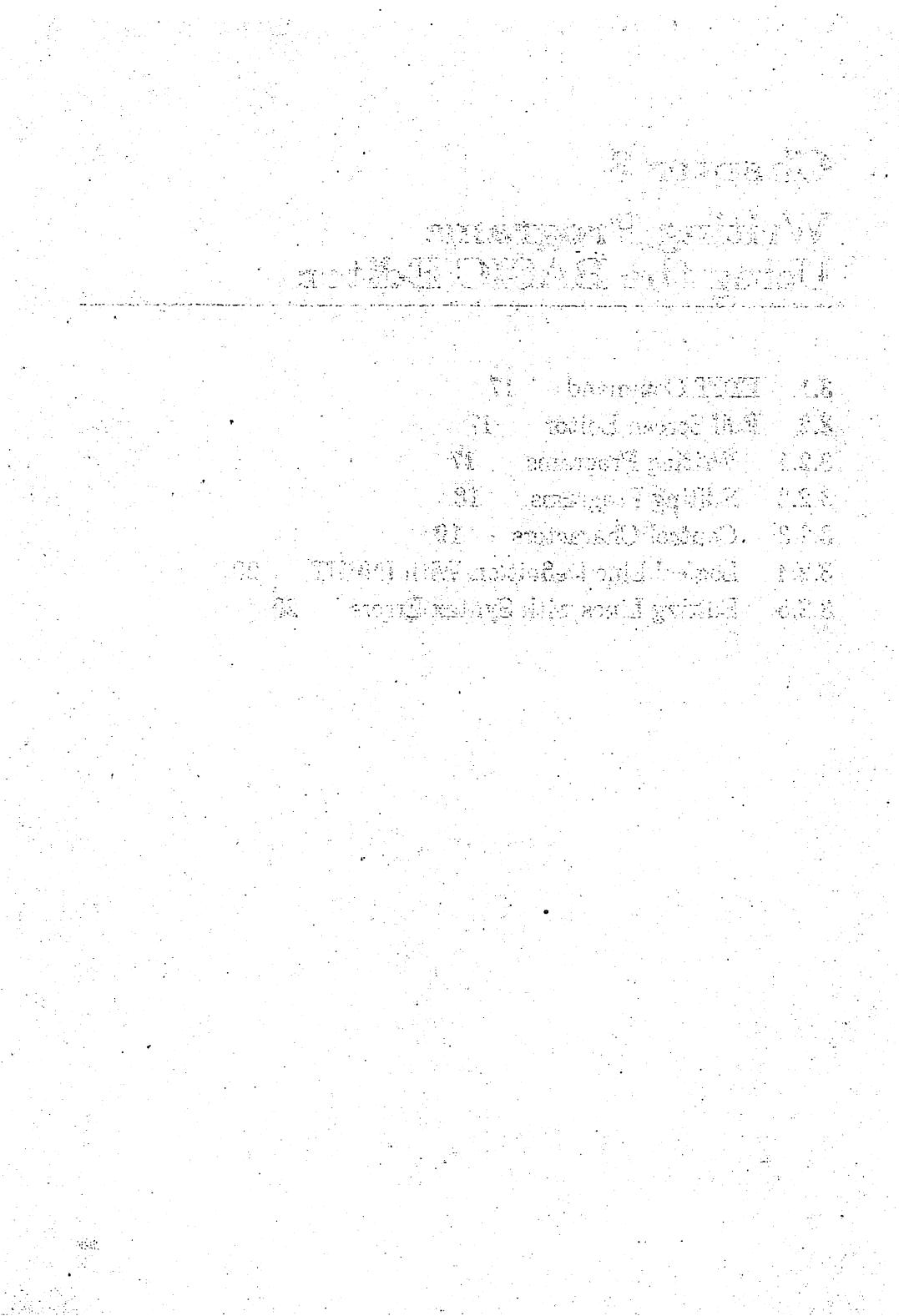
BASIC provides two ways to enter and edit text: you can issue an **EDIT** command to place you in edit mode or use the full screen editor.

3.1   EDIT Command

The **EDIT** command places the cursor on a specified line so that changes can be made to the line

3.2 Full Screen Editor

The full screen editor gives you immediate visual feedback, so that program text is entered in a "what you see is what you get" manner. If the user has a program listing on the screen, the cursor can be moved to a program line, the line edited, and the change entered by pressing the return key. This time-saving capability is made possible by special keys for cursor move-ment, character insertion and deletion, and line or screen erasure. Specific functions and key assignments are discussed in the following sections.

With the full screen editor, you can move quickly around the screen, mak-ing corrections where necessary. The changes are entered by placing the cursor on the changed line and pressing RETURN.

When input processes are directed from the screen, the user may use the full-screen editor features in responding to **INPUT** and **LINE INPUT** statements.

3.2.1   Writing Programs

You are using the full screen editor any time between the interpreter's "OK" prompt and the execution of a **RUN** command. Any line of text that is entered is processed by the editor. Any line of text that begins with a number is considered a program statement.

It is possible to extend a logical line over more than one physical line by continuing typing beyond the last column of the screen. The editor wraps the logical line so that it continues on the next physical line. A carriage re-turn signals the end of the logical line; when a carriage return is entered, the entire logical line is passed to BASIC. Up to 255 characters may be present in one logical line.

17

Program statements are processed by the editor in one of the following ways:

1. A new line is added to the program. This occurs if the line number is valid (0 through 65529) and at least one non-blank character follows the line number.

2. An existing line is modified. This occurs if the line number matches that of an existing line in the program. The existing line is replaced with the text of the new line.

3. An existing line is deleted. This occurs if the line contains only the line number, and the number matches that of an existing line.

4. The statements are passed to the command scanner for interpretation (i.e., the statement is executed).

5. An error is produced.

   If an attempt is made to delete a non-existent line, an "Undefined line" error message is displayed.

   If program memory is exhausted, and a line is added to the program, an "Out of memory" error message is displayed, and the line is not added.

More than one statement may be placed on a line. If this is done, the statements must be separated by a colon (:). The colon need not be surrounded by spaces.

3.2.2 Editing Programs

Use the **LIST** command to display an entire program or range of lines on the screen so that they can be edited with the full screen editor. Text can then be modified by moving the cursor to the place where the change is needed and then performing one of the following actions:

1. Typing over existing characters

2. Deleting characters to the right of the cursor

3. Deleting characters to the left of the cursor

4. Inserting characters

5. Appending characters to the end of the logical line

These actions are performed by special keys assigned to the various full screen editor functions (see the next section).

Changes to a line are recorded when a carriage return is entered while the cursor is somewhere on that line. The carriage return enters all changes for that logical line, and, up to the 255 character line limitation, no matter how many physical lines are included and no matter where the cursor is located on the line.

3.2.3   Control Characters

Table 3-1 lists the hexadecimal codes for the BASIC control characters and summarizes their functions. The control key combination normally assigned to each function is also listed.

Individual control functions are described following Table 3-1.

Table 3.1

BASIC Control Functions

Hex. Code	Control Key	Function
01	CONTROL-A	Enter edit mode
02	CONTROL-B	Move cursor to start of previous word
03	CONTROL-C	Break
04	CONTROL-D	Ignored
05	CONTROL-E	Truncate line (clear text to end of logical line)
06	CONTROL-F	Move cursor to start of next word
07	CONTROL-G	Beep
08	CONTROL-H	Backspace, deleting characters passed over
09	CONTROL-I	Tab (8 spaces)
0A	CONTROL-J	Linefeed
0B	CONTROL-K	Move cursor to home position
0C	CONTROL-L	Clear window
0D	CONTROL-M	Carriage return (enter current logical line)
0E	CONTROL-N	Append to end of line
0F	CONTROL-O	Suspend or restart program output
10	CONTROL-P	Ignored
11	CONTROL-Q	Restart suspended program
12	CONTROL-R	Toggle insert/typeover mode
13	CONTROL-S	Suspend program
14	CONTROL-T	Display function key contents
15	CONTROL-U	Clear logical line
16	CONTROL-V	Ignored
17	CONTROL-W	Delete word
18	CONTROL-X	Display previous program line.

19	CONTROL-Y	Display following program line.
1A	CONTROL-Z	Clear to end of window
1B	CONTROL-|	Ignored or start of control sequence (GW-BASIC option)
1C	CONTROL-e	Cursor right
1D	CONTROL-|	Cursor left
1E	CONTROL-^	Cursor up
1F	CONTROL-_	Cursor down (underscore)
7F	CONTROL-DEL	Delete character at cursor

3.2.4 Logical Line Definition With INPUT

Normally, a logical line consists of all the characters on each of the physical lines that make up the logical line. During execution of an **INPUT** or **LINE INPUT** statement, however, this definition is modified slightly to allow for forms input. When either of these statements is executed, the logical line is restricted to characters actually typed or passed over by the cursor. The insert and delete modes move only characters that are within that logical line, and delete mode will decrement the size of the line.

Insert mode increments the logical line except when the characters moved will write over non-blank characters that are on the same physical line but not part of the logical line. In this case, the non-blank characters *not* part of the logical line are preserved and the characters at the end of the logical line are thrown out. This preserves labels that existed prior to the **INPUT** statement.

3.2.5 Editing Lines with Syntax Errors

When a syntax error is encountered during program execution, BASIC prints the line containing the error and enters direct mode. You can correct the error, enter the change, and reexecute the program. When a line is modified, all files are closed, and all variables are lost. Thus, if the user wishes to examine the contents of variables just before the syntax error was encountered, the user should print the values before modifying the program line. Alternative ways to get to direct mode without erasing variable values or closing files are the **STOP** and **END** commands.

Chapter 4
Working With Files and Devices

This chapter discusses the way files and devices are used and addressed in GW-BASIC, and the way information is input and output through the system.

4.1 Default Device

When a file specification is given (in commands or statements such as FILES, OPEN, KILL), the default (current) disk drive is the current operating system default drive at the time the interpreter was invoked.

4.2 Device-Independent Input/Output

Microsoft GW-BASIC provides device-independent input/output that permits flexible approaches to data processing. Using device independent I/O means that the syntax for access is the same for any device.

The following statements, commands, and functions support device-independent I/O:

CHAIN	OPEN
CLOSE	POS
EOF	PRINT
GET	PRINT USING
INPUT	PUT
INPUT$	RUN
LINE INPUT	SAVE
LOC	WIDTH
LOF	WRITE
MERGE	

4.3 File Names and Paths

GW-BASIC uses DOS enhanced directory naming conventions, allowing files to be accessed through their path name.

4.3.1 File Name Specifications

File specifications may include path specifications. All file specifications may begin with a device specification such as **A:** or **B:** or **COM1:** or **LPT1:**. If no device is specified, the current drive is assumed. For example:

```
RUN "NEWFILE.EXE"
RUN "A:NEWFILE.EXE"
RUN "KYBD:NEWFILE.EXE"
```

4.3.2 Path Names

A path name is a sequence of directory names followed by a simple file name, each separated from the previous one by a backslash (\), and no longer than 128 characters. If a device is specified, it must be specified at the beginning of the path name. A simple file name is a sequence of characters that can optionally be preceded by a drive designation, be devoid of backslashes, and be optionally followed by an extension. The syntax of path names is:

⟦d:⟧⟦directory⟧\ ⟦directory...⟧\ ⟦filename⟧

If a path name begins with a backslash, DOS searches for the file beginning at the root directory. Otherwise, DOS begins at the user's current directory, known as the working directory, and searches downward from there.

When you are in your working directory, a file name and its corresponding path name may be used interchangeably. Some sample names are:

\	Indicates the root directory.
\PROGRAMS	Sample directory under the root directory containing program files.

\USER\MARY\FORMS\1A

> A typical full path name. This one happens to be a file named 1A in the directory named FORMS belonging to the a subdirectory of USER named MARY.

USER\SUE

> A relative path name; it names the file or directory SUE in subdirectory USER of the working directory. If the working directory is the root (\), (\), it names \USER\SUE.

TEXT.TXT

> Name of a file or directory in the working directory.

DOS provides special shorthand notations for the *working* directory and the *parent* directory (one level up) of the working directory:

> DOS uses this shorthand notation to indicate the name of the working directory in all hierarchical directory listings. DOS automatically creates this entry when a directory is made.

..
> The shorthand name of the working directory's parent directory. If you type:

```
DIR ..
```

> then DOS will list the files in the parent directory of your working directory. If you type:

```
DIR ..\..
```

> then DOS will list the files in the parent's PARENT directory.

4.3.3 Working with Path Names in BASIC

BASIC can create, change, and remove paths, with the commands **MKDIR, CHDIR**, and **RMDIR**.

The following BASIC statement creates a new directory, ACCOUNTS, in the working directory of the current drive:

```
MKDIR "ACCOUNTS"
```

The following BASIC statement changes the current directory on B: to EXPENSES:

```
CHDIR "B:EXPENSES"
```

The following BASIC statement deletes an existing directory, CLIENTS, as long as that directory is empty of all files except "." and "..":

```
RMDIR "CLIENTS"
```

For further information on handling paths in BASIC, see the **CHDIR**, **ENVIRON**, **ENVIRON\$**, **MKDIR**, and **RMDIR** statements and functions.

4.4   Program File Commands

This section is a summary of the commands and statements used in program file manipulation. All file specifications may include the device and path name.

RUN *filespec*   Loads the program from file into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program.

CHAIN *filespec*
Passes control to the named program, and passes the use of the variables and their current values to the new program.

KILL *"filespec"*
Deletes the file from the disk. *filespec* can be a program file or a sequential or random access data file.

NAME *oldfilespec* **AS** *newfilespec*
Changes the name of a file. **NAME AS** *filespec* can be used with program files, random access files, or sequential files. Path names are not permitted.

4.5   Data Files:
Sequential and Random Access I/O

There are two types of disk data files that can be created and accessed by a BASIC program: sequential files and random access files.

4.5.1   Sequential Files

Sequential files are easier to create than random access files, but are limited in flexibility and speed when it comes to locating data. The data written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order sent. The data is read back sequentially, one item after another.

The following statements and functions are used with sequential data files in sequential order:

**OPEN
WIDTH
PRINT#
PRINT USING#
WRITE#
INPUT#
INPUT$
LINE INPUT#
EOF
LOC
LOF
CLOSE**

4.5.2   Creating a Sequential File

Program 1 is a short program that creates a sequential file, DATA, from information you input at the keyboard.

■   **Program 1--Create a Sequential Data File**

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$ = "DONE" THEN END
30 INPUT "DEPARTMENT";DEPT$
40 INPUT "DATE HIRED";HIREDATE$
50 PRINT#1,N$;",";DEPT$;",";HIREDATE$
60 PRINT
70 GOTO 20
```

When the program is run, the output might look like this:

NAME? SAMUEL GOLDWYN

```
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? MARVIN HARRIS
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? DEXTER HORTON
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/81

NAME? STEVEN SISYPHUS
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/81
```

As illustrated in Program 1, the following program steps are required to create a sequential file and access the data in it:

1. **OPEN** the file in **"O"** mode.

2. Write data to the file using the **PRINT#** statement. (**WRITE#** can be used instead.)

3. To access the data in the file, you must **CLOSE** the file and reopen it in "I" mode.

4. Use the **INPUT#** statement to read data from the sequential file into the program.

4.5.3 Reading Data From a Sequential File

Now look at Program 2. It accesses the file DATA that was created in Program 1 and displays the name of everyone hired in 1981.

■ **Program 2--Accessing a Sequential File**

```
10 OPEN "I", #1, "DATA"
20 INPUT#1,N$,DEPT$,HIREDATE$
30 IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
40 GOTO 20
```

The output might look like this:

```
DEXTER HORTON
STEVEN SISYPHUS
Input past end in 20
```

Program 2 reads, sequentially, every item in the file, and prints the names of employees hired in 1981. When all the data has been read, line 20 causes an "Input past end" error. To avoid this error, use the **WHILE...WEND** control structure, which uses the **EOF** function to test for the end-of-file. The revised program looks like:

```
10 OPEN "I", #1, "DATA"
15 WHILE NOT EOF(1)
20      INPUT#1,N$,DEPT$,HIREDATE$
30      IF RIGHT$(HIREDATE$,2) = "81" THEN PRINT N$
40 WEND
```

A program that creates a sequential file can also write formatted data to the disk with the **PRINT#** **USING** statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to the file without explicit delimiters. The commas at the end of the format string separate the items in the disk file.

If you want commas to appear in the file as delimiters between variables, the **WRITE#** statement can be used. For example, the statement

```
WRITE 1, A, B$
```

could be used to write these two variables to the file with commas delimiting them.

The **LOC** function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was opened. A sector is a 128-byte block of data.

4.5.4   Adding Data to a Sequential File

If you have a sequential file residing on disk and want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in the output ("O") mode, you destroy its current contents.

Instead, use the append "A" mode. If the file doesn't already exist, the **OPEN** statement will work exactly as it would if output ("O") mode had been specified.

The following procedure can be used to add data to an existing file called FOLKS.

■   Program 3--Adding Data to a Sequential File

```
110 OPEN "A",#1,"FOLKS"
120 REM ADD NEW ENTRIES TO FILE
130  INPUT "NAME";N$
140  IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
150  LINE INPUT "ADDRESS? ";ADDR$
160  LINE INPUT "BIRTHDAY? ";BIRTHDATE$
170  PRINT#1,N$
180  PRINT#1,ADDR$
190  PRINT#1,BIRTHDATE$
200 GOTO 120
210 CLOSE #1
```

4.5.5   Random Access Files

Creating and accessing random access files requires more program steps than creating and accessing sequential files. However, there are advantages to using random access files. One advantage is that random access files require less room on the disk, since BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage of using random access files is that data can be accessed randomly, i.e., anywhere on the disk. However, it is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, each of which is numbered.

The statements and functions that are used with random access files are:

Statements	Functions
OPEN	CVD
FIELD	CVI
GET	CVS
LOC	MKS$
LOF	MKD$

LSET   MKI$

RSET

PUT

CLOSE

4.5.6   Creating a Random Access File

■ **Program 4--Create a Random File**

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";PERSON$
50 INPUT "AMOUNT";AMOUNT
60 INPUT "PHONE";TELEPHONE$
65 PRINT
70 LSET N$=PERSON$
80 LSET A$=MKS$(AMOUNT)
90 LSET P$=TELEPHONE$
100 PUT #1,CODE%
110 GOTO 30
```

As illustrated by Program 4, the following program steps are required to create a random access file.

1. **OPEN** the file for random access (**"R"** mode). The following example specifies a record length of 32 bytes. If the record length is not specified, the default is 128 bytes. For example:

    ```
    OPEN "R",#1,"FILE",32
    ```

2. Use the **FIELD** statement to allocate space in the random buffer for the variables that will be written to the random access file. For example:

    ```
    FIELD #1, 20 AS N$, 4 AS ADDR$, 8 AS P$
    ```

3. Use **LSET** to move the data into the random access buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: **MKI$** to make an integer value into a string, **MKS$** to make a single precision value into a string, and **MKD$** to make a double precision value into a string. For example:

    ```
    LSET N$=X$
    LSET ADDR$=MKS$(AMT)
    ```

```
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the **PUT** statement. For example:

```
PUT #1,CODE%
```

Program 4 takes information that is input at the terminal and writes it to a random access file. Each time the **PUT** statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Note

Do not use a fielded string variable in an **INPUT** or **LET** statement. Doing so causes that variable to be redeclared, after which BASIC will no longer associate that variable with the file buffer, but with the new program variable.

4.5.7 Accessing a Random Access File

Program 5 accesses the random access file **FILE** that was created in Program 4. By entering a three-digit code at the keyboard terminal, the information associated with that code is read from the file and displayed.

■ **Program 5 -- Access a Random File**

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##" CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

The following program steps are required to access a random access file:

1. **OPEN** the file in "R" mode, as follows:

```
OPEN "R",#1,"FILE",32
```

2. Use the **FIELD** statement to allocate space in the random access buffer for the variables that will be read from the file. For example:

```
FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
```

Note

In a program that performs both input and output on the same random access file, you can often use just one **OPEN** statement and one **FIELD** statement.

---

3. Use the **GET** statement to move the desired record into the random access buffer:

```
GET #1, CODE%
```

4. The data in the buffer can now be accessed by the program. Numeric values that were converted to strings by the **MKS$**, **MKD$** or **MKI$** statements must be converted back to numbers using the "convert" functions: **CVI** for integers, **CVS** for single precision values, and **CVD** for double precision values. The **MKI$** and **CVI** processes mirror each other, the former converting a number into a format for storage in random files, the latter converting the random file storage into a format usable by the program. For example:

```
PRINT N$
PRINT CVS(A$)
```

The **LOC** function when used with random access files, returns the "current record number." The current record number is the last record number that was used in a **GET** or **PUT** statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends program execution if the current record number in file#1 is greater than 50.

4.5.8 Random File Operations

Program 6 is an inventory program that illustrates random file access.

■ Program 6--Inventory

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT "1,INITIALIZE FILE"
140 PRINT "2,CREATE A NEW ENTRY"
150 PRINT "3,DISPLAY INVENTORY FOR ONE PART"
160 PRINT "4,ADD TO STOCK"
170 PRINT "5,SUBTRACT FROM STOCK"
180 PRINT "6,DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION < 1) OR (FUNCTION > 6) THEN PRINT
    "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM ** BUILD NEW ENTRY **
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT "OVERWRITE"; ADDR$:
    IF ADDR$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESCRIPTION$
300 LSET D$=DESCRIPTION$
310 INPUT "QUANTITY IN STOCK";QUANTITY%
320 LSET Q$=MKI$(QUANTITY%)
330 INPUT "REORDER LEVEL";REORDER%
340 LSET R$=MKI$(REORDER%)
350 INPUT "UNIT PRICE";PRICE
360 LSET P$=MKS$(PRICE)
370 PUT#1,PART%
380 RETURN
390 REM ** DISPLAY ENTRY **
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##"
CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";ADDITIONAL%
```

```
520 Q%=CVI(Q$)+ADDITIONAL%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";LESS%
610 Q%=CVI(Q$)
620 IF (Q%-LESS%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-LESS%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
        ." REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710    GET#1,I
720  ' IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
        CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
        GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";CONFIRM$:IF CONFIRM$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940    PUT#1,I
950 NEXT I
960 RETURN
```

In this program, the record number is used as the part number. It is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing

CHR$(255)

as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the various inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

4.6 BASIC and Child Processes

Through the use of the **SHELL** statement, GW-BASIC is able to use one of
the most powerful features of DOS: the ability to create child processes.
SHELL enables the user to run part of a BASIC program, temporarily exit
to DOS to perform a specified function, and return to the BASIC program
at the statement after the **SHELL** statement to proceed with the rest of
the program.

BASIC will produce a child program when it uses the **SHELL** statement.
It is not possible for BASIC to totally protect itself from its children. When
a **SHELL** statement is executed, many things may be going on. For exam-
ple, files may be **OPEN** and devices may be in use.

Chapter 5
Using Advanced Features

5.1 Assembly Language Subroutines

You may call assembly language subroutines from your BASIC program with the USR function or the CALL or CALLS statement.

It is recommended that you use the CALL or CALLS statement for interfacing 8086 machine language programs with BASIC. These statements are more readable and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than its alternative, the USR function.

5.1.1 Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded. To do so, use the /M: switch during start-up. The /M: option sets the highest memory location to be used by BASIC.

In addition to the BASIC code area, BASIC uses up to 64K of memory beginning at its data (DS) segment.

If more stack space is needed when an assembly language subroutine is called, you can save the BASIC stack and set up a new stack for use by the assembly language subroutine. The BASIC stack must be restored, however, before you return from the subroutine.

The assembly language subroutine can be loaded into memory in several ways, the most simple being to use the BLOAD command. Also, the user could SHELL a program that exits, but stays resident, leaving the linked, relocated image in memory. As a third choice, the user could execute a program that exits but stays resident, and then run BASIC.

The following guidelines must be observed if you choose to BLOAD, or read and poke, an EXE file into memory:

1. Make sure the subroutines do not contain any long references, address offsets that exceed 64K or that take the user out of the code segment. These long references require handling by the EXE loader.

2. Skip over the first 512 bytes (the header) of the linker's output file (EXE), then read in the rest of the file.

5.1.2 Internal Representation

The following section describes the internal representation of numbers in BASIC. Knowledge of these arrangements is critical for many assembly language programming routines.

Single Precision - 24 bit mantissa

```
|   O   |   1   |   2   |   3   |
---------------------------------
| loman |       |S|himan|  exp  |
---------------------------------
```
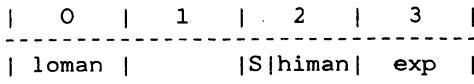
where loman = the low mantissa
      S = the sign
      himan = the high mantissa
      exp = the exponent
      man = himan:...:loman

— If $exp = 0$, then $number = 0$.

— If $exp <> 0$, then the mantissa is normalized and

$$<number> = <sgn> * .1<man> * 2 ** (<exp>-80h)$$

That is, in single precision (hex notation - bytes low to high)

```
00000080    =    .5
00008080    =   -.5
```

Double Precision - 56 bit mantissa

```
|   O   |   1   |   2   |   3   |   4   |   5   |   6   |   7   |
---------------------------------------------------------------
| loman |       |       |       |       |       |S|himan|  exp  |
---------------------------------------------------------------
```
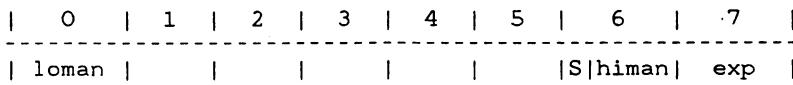
5.1.3 CALL Statement

The **CALL** statement is the recommended way of interfacing 8086 machine language subroutines with BASIC. Do not use the **USR** function unless you are running previously written subroutines that already contain **USR** functions.

The syntax of the **CALL** statement is:

CALL *variablename* [[(*argumentlist*)]]

where *variablename* contains the offset into the current segment that is the starting point in memory of the subroutine being called. The current segment is either the default, or that which has been defined by a **DEF SEG** statement.

The *argumentlist* contains the variables or constants, separated by commas, that are to be passed to the subroutine.

Invoking the **CALL** statement causes the following to occur:

1. For each argument in the argument list, the two-byte offset of the argument's location within the BASIC segment is pushed onto the stack.

2. Control is transferred to the subroutine with an 8086 long call to the segment address given in the last **DEF SEG** statement and the offset given in *variablename*.

Figures 6.1 and 6.2 illustrate the state of the stack at the time the **CALL** statement is executed, and the condition of the stack during execution of the called subroutine, respectively.

```
                  +-----------------------+
                  |     argument 0        |
   high           +-----------------------+SP+4+(2*n)
  addresses       |     .                 |
     |            |     .                 |
     |            |     .                 |       Each argument is a 2-byte
     |            +-----------------------+       pointer into memory
    |c            |     argument n-1      |
   s|o            +-----------------------+SP+6
   t|u            |     argument n        |
   a|n            +-----------------------+SP+4
   c|t            | return segment address|
   k|e            +-----------------------+SP+2
    |r            | return offset         |
     |            +-----------------------+SP     <--stack pointer
     V            |                       |          (SP register
   low            |                       |           contents)
  addresses       |                       |
```
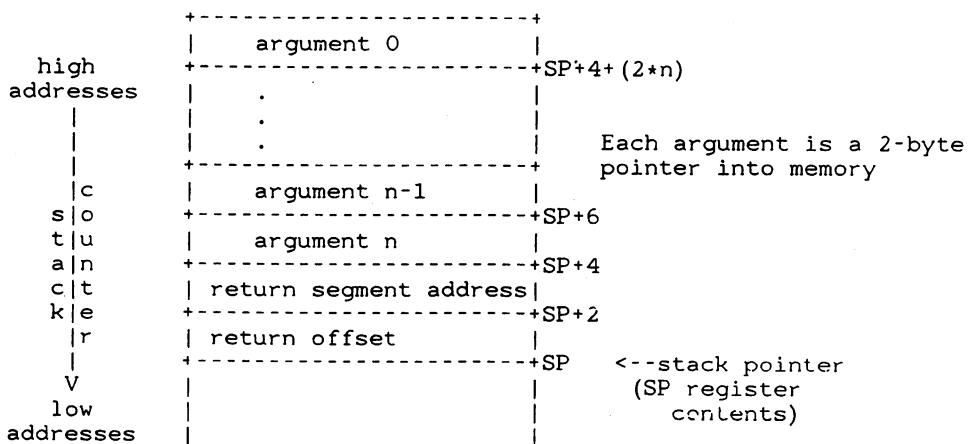
Figure 5.1 Stack layout when CALL statement is activated

After the CALL statement has been activated, the subroutine has control. Arguments may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

```
high         |----------------------|
addresses    |                      |\
       |     |   argument 0         | \
       |     |   argument 1         |  \
       |  `  |   .                  |   }<--Absent if any argument
       |     |   .                  |  /     referenced within a ne
    |c       |   .                  | /      procedure
  s |o       |   argument n         |/
  t |u       |                      |
  a |n       +----------------------+
  c |t       | return segment address|}<--Absent in local procedure
  k |e       +----------------------+
    |r       | return offset        |
    |        +-----+----------------+  <--stack pointer
    |        |                      |           (SP register
    |        |                      |              contents)
    |        |                      |
    |        |   local variables    |
    |        |   (data pushed on    |
    |        |    stack)            |
    |        |    .                 |
    |        |    .                 |
    |        |    .                 |
    |        |                      |
    |        +----------------------+
    |        |   This space may be  |
    |        |   used during pro-   |   Stack pointer may change
    |        |   cedure execution   |    during procedure executi
    |        |    .                 |
    |        |    .                 |
    |        |    .                 |
    |        |                      |
    V        |                      |
  low        |                      |
addresses    |                      |
```

Figure 5.2 Stack layout during execution of a CALL statement

Observe the following rules when coding a subroutine:

1. The called routine must preserve segment registers DS, ES, SS, and the base pointer (BP). If interrupts are disabled in the routine, they must be enabled before exiting. The stack must be cleaned up on exit.

2. The called program must know the number and length of the arguments passed. The following routine shows an easy way to reference arguments:

```
PUSH    BP
MOV     BP,SP
ADD     BP, (2*number of arguments)+4
```

   Then:

   argument 0 is at BP
   argument 1 is at BP-2
   argument n is at BP-2*n

   (number of arguments = n+1)

3. Variables may be allocated either in the code segment or on the stack. Be careful not to modify the return segment and offset stored on the stack.

4. The called subroutine must clean up the stack. A preferred way to do this is to perform **RET** *n* statement (where *n* is two times the number of arguments in the argument list) to adjust the stack to the start of the calling sequence.

5. Values are returned to BASIC by including in the argument list the name of the variable that will receive the result. The internal format for numbers in GW-BASIC is discussed earlier in this chapter.

6. If the argument is a string, the argument's offset points to 3 bytes which, as a unit, are called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

Warning

   If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add

```
+""
```

to the string literal in the program. For example, use

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Then the string may be modified without affecting the program.

---

7. The contents of a string may be altered by user routines, but the descriptor *must not* be changed. Do not write past the end-of-string. GW-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

8. Data areas needed by the routine must be allocated either in the CODE segment of the user routine or on the stack. It is not possible to declare a separate data area in the user assembler routine.

Example of **CALL** statement:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO(A,B$,C)
   .
   .
   .
```

Line 100 sets the segment to 8000 Hex. The value of variable FOO is added into the address as the low word after the **DEF SEG** value is left shifted 4 bits. Here, the long call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence in 8086 assembly language demonstrates access to the arguments passed. The returned result is stored in the variable C.

```
PUSH    BP          ;Set up pointer to arguments
MOV     BP,SP
ADD     BP,(4+2*3)
MOV     BX,[BP-2]   ;Get address of B$ descriptor.
MOV     CL,[BX]     ;Get length of B$ in CL.
MOV     DX,1[BX]    ;Get addr of B$ text in DX.
        .
        .
        .
MOV     SI,[BP]     ;Get address of 'A' in SI.
MOV     DI[BP-4]    ;Get pointer to 'C' in DI.
MOVS    WORD        ;Store variable 'A' in 'C'.
```

```
POP    BP          ;Restore pointer.
RET    6           ;Restore stack, return.
```

Important

> The called program must know the variable type for the numeric arguments passed. In the previous example, the instruction
>
> ```
> MOVS WORD
> ```
>
> will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision format and copy 8 bytes if they were double precision.

5.1.4 CALLS Statement

The **CALLS** statement should be used to access subroutines that were written using Microsoft FORTRAN calling conventions. **CALLS** works just like **CALL**, except that with **CALLS** the arguments are passed as segmented addresses, rather than as unsegmented addresses.

Because MS-FORTRAN routines need to know the segment value for each argument passed, the segment is pushed and then the offset is also pushed. For each argument, four bytes are pushed rather than 2, as in the **CALL** statement. Therefore, if your assembler routine uses the **CALLS** statement, *n* in the **RET** *n* statement is four times the number of arguments.

5.1.5 USR Function

Although using the **CALL** statement is the recommended way of calling assembly language subroutines, the **USR** function is also available for this purpose. This ensures compatibility with older programs that contain **USR** functions.

USR[[*digit*]][[(*argument*)]]

where *digit* is from 0 to 9. The *digit* specifies which **USR** routine is being called. If *digit* is omitted, **USR0** is assumed.

45

The *argument* is any numeric or string expression. Arguments are discussed in detail in the following paragraphs.

In the BASIC Interpreter, a **DEF SEG** statement *must* be executed prior to a **USR** function call to assure that the code segment points to the subroutine being called. The segment address given in the **DEF SEG** statement determines the starting segment of the subroutine.

For each **USR** function, a corresponding **DEF USR** statement must be executed to define the **USR** function call offset. This offset and the currently active **DEF SEG** address determine the starting address of the subroutine.

When the **USR** function call is made, register **AL** contains a value that specifies the type of argument that was given. The value in **AL** may be one of the following:

Value in AL	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a number, the **BX** register points to the Floating-Point Accumulator (**FAC**) where the argument is stored.

If the argument is an integer:

FAC-2 contains the upper 8 bits of the integer.
FAC-3 contains the lower 8 bits of the integer.

For versions of GW-BASIC that use binary floating-point:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive, 1 = negative).

If the argument is a single precision floating-point number:

FAC-2
contains the middle 8 bits of mantissa.
FAC-3
contains the lowest 8 bits of mantissa.

46

If the argument is a double precision floating-point number:

FAC-7 through **FAC-4** contain four more bytes of mantissa (**FAC-7** contains the lowest 8 bits).

If the argument is a string, the **DX** register points to 3 bytes which, as a unit, are called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in the BASIC data segment.

Warning

* If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy the program this way.

Usually, the value returned by a **USR** function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

BASIC has extended the **USR** function interface to allow calls to **MAKINT** and **FRCINT**. This allows access to these routines without giving their absolute addresses. The address **ES:BP** is used as an indirect far pointer to the routines **FRCINT** and **MAKINT**.

To call **FRCINT** from a **USR** routine use

```
CALL DWORD ES:[BP]
```

To call **MAKINT** from a **USR** routine use

```
CALL DWORD ES:[BP+4]
```

Example

```
110 DEF USR0=&H8000 'Assumes decimal argument /M:32767
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must

47

be consistent with that of the argument used.

5.2   Event Trapping

Event trapping allows a program to transfer control to a specific program
line when a certain event occurs. Control is transferred as if a **GOSUB**
statement had been executed to the trap routine starting at the specified
line number. The trap routine, after servicing the event, executes a
RETURN statement that causes the program to resume execution at the
place where it was when the event trap occurred.

The events that can be trapped are receipt of characters from a communi-
cations port (**ON COM**), detection of certain keystrokes (**ON KEY**), time
passage (**ON TIMER**), emptying of the background music queue (**ON
PLAY**), joystick trigger activation (**ON STRIG**), and lightpen activation
(**ON PEN**).

This section gives an overview of event trapping. For more details on indi-
vidual statements, see individual reference pages.

Event trapping is controlled by the following statements:

eventspecifier **ON**   Turn on trapping
eventspecifier **OFF** Turn off trapping
eventspecifier **STOP** Temporarily turns off trapping

where *eventspecifier* is one of the following:

COM (*n*)   where *n* is the number of the communications chan-
                         nel. The *n* is the same device referred to in a
                         COM*n*: statement. The **COM** channels are num-
                         bered 1 through *n*, where *n* is implementation
                         dependent.

                         Typically, the **COM** trap routine will read an
                         entire message from the **COM** port before return-
                         ing. The **COM** trap is not recommended for single
                         character messages because at high baud rates the
                         overhead of trapping and reading for each charac-
                         ter may allow the interrupt buffer for **COM** to
                         overflow.

48

KEY *n*   where *n* is a trappable key number. Trappable keys are the function and cursor keys, and any user defined keys. See the "KEY ON Statement" for definitions of key numbers.

Note that KEY *(n)* ON is not the same statement as KEY ON. KEY *(n)* ON sets an event trap for the specified key. displays the values of all the function keys on the twenty-fifth line of the screen.

When the GW-BASIC Interpreter is in direct mode function keys maintain their standard meanings.

When a key is trapped, that occurrence of the key is destroyed. Thuys, you cannot subsequently use the INPUT or INKEY$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

PEN   Since there is only one lightpen, no number is given when PEN trapping is enabled.

TIMER   ON TIMER(n), where n is a numeric expression representing a number of seconds since the previous midnight. The ON TIMER statement can be used to perform background tasks at defined intervals.

PLAY   ON PLAY(n), where n is a number of notes left in the music buffer. The ON PLAY statement is used to retrieve more notes from the background music queue, to permit continuous background music during program execution.

STRIG *n*   where *n* is the number of the joystick trigger. The range for *n* is 0 to 2.

5.2.1   ON GOSUB Statement

The ON GOSUB statement sets up a line number for the specified event trap. The format is:

ON *eventspecifier* GOSUB *linenumber*

A *linenumber* of zero disables trapping for that event.

When an event is **ON** and if a non-zero line number has been specified in the **ON GOSUB** statement, every time BASIC starts a new statement it will check to see if the specified event has occurred (e.g., the lightpen has been struck or a COM character has come in). When an event is **OFF**, no trapping takes place, and the event is not remembered even if it takes place.

When an event is stopped (i.e., when an *eventspecifier* **STOP** statement), no trapping takes place, but the occurrence of an event is remembered so that an immediate trap will take place when an *eventspecifier* **ON** statement is executed.

When a trap is made for a particular event, the trap automatically causes a **STOP** on that event, so recursive traps can never occur. A return from the trap routine automatically executes an **ON** statement unless an explicit **OFF** has been performed inside the trap routine.

Note that once an error trap takes place, all trapping is automatically disabled. In addition, event trapping will never occur when BASIC is not executing a program.

5.2.2 RETURN Statement

When an event trap is in effect, a **GOSUB**]statement will be executed as soon as the specified event occurs. For example, the statement

```
ON PEN GOSUB 1000
```

specifies that the program go to line 1000 as soon as the pen is used. If a simple **RETURN** statement is executed at the end of this subroutine, program control will return to the statement following the one where the trap occurred. When the **RETURN** statement is executed, its corresponding **GOSUB** return address is canceled.

GW-BASIC includes the **RETURN** *linenumber* enhancement, which lets processing resume at a definable line. Normally, the program returns to the statement immediately following the **GOSUB** statement when the **RETURN** statement is encountered. However, **RETURN** *linenumber* enables the user to specify another line. If not used with care, however, this capability may cause problems. Assume, for example, that your program contains:

```
10 ON PEN GOSUB 1000
20 FOR I = 1 TO 10
30 PRINT I
```

```
40 NEXT I
50 REM NEXT PROGRAM LINE

200 REM PROGRAM RESUMES HERE

1000 'FIRST LINE OF SUBROUTINE
  .
  .
  .
1050 RETURN 200
```

If the pen is activated while the **FOR/NEXT** loop is executing, the subroutine will be performed, but program control will return to line 200 instead of completing the **FOR/NEXT** loop. The original **GOSUB** entry will be canceled by the **RETURN** statement, and any other **GOSUB**, **WHILE**, or **FOR** (e.g., an **ON STRIG** statement) that was active at the time of the trap will remain active. But the current **FOR** context will also remain active, and a "FOR without NEXT" error may result.

Chapter 6
Language Reference

This chapter presents the character set and the rules for the constants, variables, expressions, and operators used by the Microsoft GW-BASIC language. It also includes an alphabetic reference to the statements and functions in the GW-BASIC language.

6.1 Character Set

The Microsoft GW-BASIC character set consists of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in GW-BASIC are the uppercase and lowercase letters of the English alphabet.

The GW-BASIC numeric characters are the digits 0 through 9. The alphabetic characters A, B, C, D, E, and F may be used as part of hexadecimal numbers.

Special Characters

The following special characters and terminal keys are recognized by GW-BASIC:

Character	Name or Function
	Blank
=	Equal sign or assignment symbol
+	Plus sign
—	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent

#	•Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At symbol
_	Underscore
RETURN	Terminates input of a line

6.2 Constants

Constants are the actual values GW-BASIC uses during program execution.
There are two types of constants: string and numeric. A string constant is
a sequence of up to 255 alphanumeric characters enclosed in double quota-
tion marks. These are all valid string constants:

```
"HELLO"
"$25,000.000"
"Number of Employees"
```

Numeric constants are positive or negative numbers. There are five types of numeric constants:

- Integer constants

    Whole numbers between –32,768 and +32,767. Integer constants do not contain decimal points.

- Fixed-point constants

    Positive or negative real numbers; that is, numbers that contain decimal points.

- Floating-point constants

    Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is $1.18 * 10^{-38}$ to $3.4 * 10^{+38}$ in the binary math package, and 10^{-64} to 10^{+63} in the decimal math package. (Double precision floating-point constants are denoted by the letter D instead of E.)

    Examples:

    $235.988E-7 = .0000235988$
    $2359E6 = 2359000000$

- Hex constants

    Hexadecimal numbers with the prefix &H.

    Examples:

    &H76
    &H32F

- Octal constants

    Octal numbers with the prefix &O or &.

    Examples:

    &O347
    &1234

Numeric constants can be either single precision or double precision numbers. Single precision numeric constants are stored with 6 digits of precision (plus the exponent) and printed with up to 6 digits of precision. Double precision numbers are stored with 14 digits of precision (plus the exponent) and printed with up to 14 digits of precision. A single precision

constant is any numeric constant that has one of the following properties:

- Six or fewer digits
- Exponential form denoted by E
- A trailing exclamation point (!)

A double precision constant is any numeric constant that has one of the following properties:

- Seven or more digits
- Exponential form denoted by D
- A trailing number sign (#)

The following are examples of numeric constants:

Single Precision	Double Precision
46.8	345692811
−1.09E-6	−1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Numeric constants in GW-BASIC cannot contain commas.

6.3 Variables

Variables represent values that are used in a program. As with constants, there are two types of variables: numeric and string. A numeric variable can be assigned only a number value. A string variable can be assigned only a character string value. You can assign the value of a variable, or it can be assigned as the result of calculations in the program. In either case, the variable must always match the type of data assigned to it. Before a variable is assigned a value, its value is assumed to be zero (for numeric variables) or null (for string variables).

6.3.1   Variable Names

A GW-BASIC variable name can contain as many as 40 characters. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in a variable name must be a letter. Special type declaration characters are also allowed (see the following section for more information). If a variable begins with **FN**, it is assumed to be a call to a user-defined function that has been defined with the **DEF FN** statement. A variable name cannot be a reserved word, but embedded reserved words are allowed.

For example

```
10 LOG = 8
```

is illegal because **LOG** is a reserved word. Reserved words include all GW-BASIC commands, statements, function names, and operator names (see your compiler user's guide for a complete list of these words).

6.3.2   Declaring Variable Types

Variable names can be declared as either numeric values or string values. String variable names are written with a dollar sign ($) as the last character. For example,

```
A$ = "SALES REPORT"
```

The dollar sign is the type declaration character for string variables; that is, it "declares" that the variable will represent a string. Numeric variable names can declare integer, single precision, or double precision values. Computations with integer and single precision variables are less accurate than those with double precision variables. However, you may want to declare a variable to a lower precision type because:

- Variables of higher precision take up more memory space.

- Arithmetic computation times are longer for higher precision numbers. A program with repeated calculations runs faster with integer variables.

The default type for a numeric variable is single precision.

The type declaration characters for numeric variables and the memory requirements (in bytes) for storing each variable type are listed in Table 1.1.

Table 6.1

Variable Type Memory Requirements

Declaration Character	Variable Type	Bytes Required
%	Integer	2
!	Single precision	4
#	Double precision	8
$	String	4 bytes overhead plus the present contents of the string

The following are examples of GW-BASIC variable names:

PI#	A double precision variable
MINIMUM!	A single precision variable
LIMIT%	An integer variable
FIRSTNAME$	A string variable
ABC	A single precision variable

The GW-BASIC statements **DEFINT, DEFSTR, DEFSNG,** and **DEFDBL** can be included in a program to declare the types for certain variable names. By using one of the **DEF** *type* statements, you can specify that all variables starting with a given letter will be of a certain variable type, without the trailing declaration character.

6.3.3 Array Variables

An array is a group or table of values referred to by the same variable name. The individual values in an array are called elements. Array elements are variables also. They can be used in any GW-BASIC statement or function which uses variables. Declaring the name and type of an array and setting the number of elements in the array is known as *dimensioning*

the array.

Each element in an array is referred to by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) refers to a value in a one-dimensional array, while T$(1,4) refers to a value in a two-dimensional string array. Note that the array variable T and the simple variable T are not the same variable. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,768. The maximum amount of space that may be taken for an array is 64K.

Array elements, like numeric variables, require a certain amount of memory space, depending on the variable type. See Table 1.1 for information on the memory requirements for storing arrays.

6.4   Expressions and Operators

An expression can be a string or numeric constant, a variable, or a single value obtained by combining constants, variables, and other expressions with operators. Operators perform mathematical or logical operations on values. The operators provided by GW-BASIC can be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

6.4.1   Hierarchy of Operations

The GW-BASIC operators have an order of precedence; that is, when several operations take place within the same program statement, certain kinds of operations will be executed before others. If the operations are of the same level, the leftmost one is executed first, the rightmost last. Operations are executed in the following order:

1. Exponentiation

2. Negation

3. Multiplication and Division

4. Integer Division

5. Modulo Arithmetic

6. Addition and Subtraction

7. Relational Operators

8. NOT

9. AND

10. OR and XOR

11. EQV

12. IMP

6.4.2 Arithmetic Operators

Use parentheses to change the order in which arithmetic operations are performed. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained. Here are some sample algebraic expressions and their GW-BASIC counterparts:

Algebraic Expression	BASIC Expression
$\dfrac{X-Z}{Y}$	(X-Y)/Z
$\dfrac{XY}{Z}$	X*Y/Z
$\dfrac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)

$$X(-Y) \qquad\qquad X*(-Y)$$

Two consecutive operators must be separated by parentheses.

Integer Division

Integer division is denoted by the backslash (\) instead of the slash (/), which indicates floating-point division. The operands are rounded to integers (and must be in the range –32,768 to +32,767) before the division is performed, and the quotient is truncated to an integer. For example,

```
10 X = 10\4
15 X2 = 10/4
20 Y = 25.68\6.99
25 Y2 = 25.68/6.99
30 PRINT X, X2, Y, Y2
```

Output:

```
2          2.5          3          3.6738197424893
```

Modulo Arithmetic

Modulo arithmetic is denoted by the modulus operator **MOD.** Modulo arithmetic provides the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2      (10\4=2 with a remainder 2)
25.68 MOD 6.99 = 5  (26\7=3 with a remainder 5)
```

Overflow and Division by Zero

If a division by zero is encountered during the evaluation of an expression, the "Division by zero." error message is displayed. Machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is again displayed. Positive machine infinity (the highest number the computer can produce) is supplied as the result of the exponentiation, and execution continues. If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as a result, and execution continues.

6.5 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (non-zero) or "false" (0). This result can then be used to make a decision regarding program flow.

Table 6.2

Relational Operators and Their Functions

Operator	Relation Tested	Expression
=	Equality†	X=Y
< >	Inequality	X< >Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

†The equal sign is also used to assign a value to a variable.

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first. For example, the expression

```
X + Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T–1 divided by Z.

6.6 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations, just as the relational operators can connect two or more relations and return a true or false value to be used in making a decision. For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

A logical operator returns a result from the combination of true-false operands. The result (in bits) is either "true" (-1) or "false" (zero).

There are six logical operators in GW-BASIC:

NOT	(logical complement)
AND	(conjunction)
OR	(disjunction)
XOR	(exclusive or)
IMP	(implication)
EQV	(equivalence)

Each operator returns results as indicated in Table 1.3. A "T" indicates a true value and a "F" indicates a false value. Operators are listed in order of operator precedence.

Table 6.3

Results Returned by Logical Operations

Operation	Value	Value	Result
NOT	X		NOT X
	T		F
	F		T
AND	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F
XOR	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F
EQV	X	Y	X EQV Y
	T	T	T
	T	F	F
	F	T	F
	F	F	T

Table 6.3 *(continued)*

Operation	Value	Value	Result
IMP	X	Y	X IMP Y
	T	T	T
	T	F	F
	F	T	T
	F	F	T

In an expression, logical operations are performed after arithmetic and rela-
tional operations. Logical operators convert their operands to 16-bit,
signed, two's complement integers in the range − 32,768 to +32,767. (If the
operands are not in this range, an error results.) If both operands are sup-
plied as 0 or − 1, logical operators return 0 or − 1, respectively. The given
operation is performed on these integers in bits; that is, each bit of the
result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit
pattern. For instance, the **AND** operator can be used to mask all but one
of the bits of a status byte. The **OR** operator can be used to merge two
bytes to create a particular binary value. The following examples demon-
strate how the logical operators work:

63 AND 16 = 16   63 = binary 111111 and 16 = binary 10000, so
                    63 AND 16 = 16.

15 AND 14 = 14   15 = binary 1111 and 14 = binary 1110, so 15
                    AND 14 = 14 (binary 1110).

-1 AND 8 = 8   -1 = binary 1111111111111111 and 8 = binary
                    1000, so -1 AND 8 = 8.

4 OR 2 = 6   4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6
                    (binary 110).

10 OR 10 = 10   10 = binary 1010, so 1010 OR 1010 = 1010 (10).

-1 OR -2 = -1   -1 = binary 1111111111111111 and -2 = binary
                    1111111111111110, so -1 OR -2 = -1. The bit
                    complement of 16 zeros is sixteen ones, which is
                    the two's complement representation of -1.

NOT X = –(X + 1)   The two's complement of any integer is the bit
complement plus one.

6.7 Functional Operators

A function is used in an expression to call a predetermined operation to be
performed on an operand. For example **SQR** is a functional operator used
twice in the following assignment statement:

```
A = SQR (20.25) + SQR (37)
```

GW-BASIC incorporates two kinds of functions: intrinsic and user-defined.

Intrinsic Functions

Functions perform operations on their operands and return values. GW-
BASIC has many pre-defined functions built into the language that can be
used by simply "calling" them. Examples are the **SQR** (square root) and
SIN (sine) functions.

User-Defined Functions

Functions can be defined by the user with the **DEF FN** statement. These
functions are defined only for the life of a given program, and are not part
of the GW-BASIC language. In addition to **DEF FN** compiled form of the
BASIC language supports subprograms,

6.8 String Operators

A string expression consists of string constants, string variables, and other
string expressions combined by string operators. There are two classes of
string operations: concatenation and string function.

The act of combining two strings is called concatenation. The plus symbol
(+) is the concatenation operator. For example, the following program
fragment combines the string variables A$ and B$ to produce the value
"FILENAME":

```
10 A$ = "FILE": B$ = "NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
```

Output:

```
FILENAME
NEW FILENAME
```

Strings can be compared using the same relational operators used with numbers:

$$= \ <> \ < \ > \ <= \ >=$$

String functions are similar to numeric functions, except that the operands are strings rather than numeric values. String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller if they are equal to that point. Leading and trailing blanks are significant. The following are examples of true statements:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"          (where B$ = "8/12/85")
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

6.9 Type Conversion

When necessary, GW-BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind:

- If a numeric constant of one type is set equal to a numeric variable of a different type, the numeric constant will be stored as the type declared in the variable name. (If a string variable is set equal to a

numeric value or vice versa, a "Type Mismatch" error message is
generated. For example:

```
10 A% = 23.42
20 PRINT A%
```

Output:

```
23
```

- During expression evaluation, all of the operands in an arithmetic
  or relational operation are converted to the same degree of preci-
  sion; that is, that of the most precise operand. Also, the result of
  an arithmetic operation is returned to this degree of precision. For
  example,

```
10 D# = 6/7
20 PRINT D#
```

Output:

```
.8571428571428571
```

The arithmetic operation was performed in double precision and the
result was returned in D as a double precision value.

- Logical operators convert their operands to integers and return an
  integer result. Operands must be in the range −32,768 to +32,767
  or an "Overflow" error message is generated.

- When a floating-point value is converted to an integer, the frac-
  tional portion is rounded. For example,

```
10 CAREN% = 55.88
20 PRINT CAREN%
```

Output:

```
56
```

6.10   Reference Format

Each statement or statement in the alphabetical reference that follows is
described using the following format:

Heading	**Function**
Syntax	Gives the correct syntax for the statement or function. All functions return a value of a particular type and can be used wherever an expression can be used. Unlike

functions, statements can appear alone on a BASIC program line.

Action   Summarizes what the statement or function does.

Remarks   Describes arguments and options in detail, and explains how to use the statement or function.

See Also   Cross-references related statements and functions. This is an optional section that does not appear with every reference entry.

Example   Gives sample commands, programs, and program segments that illustrate the use of the given statement or function. This is an optional section that does not appear with every reference entry.

Compiler Differences

Tells whether or not the statement or function is new in the compiler, enhanced in the compiler, or different in the compiler. This is an optional section that does not appear with every reference entry.

Note   Points out an important fact or feature. This is an optional section that does not appear with every reference entry.

Warning   Alerts the user to problems or dangers associated with the use of the given statement or function. This is an optional section that does not appear with every reference entry.

6.11   Reference Syntax Notation

The following syntax notation is used in the reference manual:

CAPS   Items in capital letters indicate BASIC keywords. These keywords must be part of the statement syntax, unless they are enclosed in brackets, in which case they are optional. In an actual program, you can enter these BASIC keywords in either uppercase letters or lowercase letters.

italics   Items in italics represent information that you must supply.

〚 〛   Items inside double square brackets are optional.

71

	Items followed by ellipses can be repeated.
...	Items followed by ellipses can be repeated.
¦ ¦	Braces indicate that you have a choice between two or more items. You must choose one of them, unless all of the items are also enclosed in square brackets.
¦	Vertical bars separate the different choices inside braces.
.	Vertical ellipses are used in syntax lines and program
.	examples to show that a portion of the program is
.	omitted.

You must enter all punctuation, including commas, parentheses, semicolons, hyphens, and equal signs, exactly as shown.

■ **Example**

In the following syntax line, "LOCK" and "TO" are BASIC keywords: you cannot leave them out of the syntax. However, you can omit the "#" symbol and everything inside the second set of brackets (provided you want to lock an entire file). If you want to lock just certain records, you have a choice: one record or a range of records TO *end*). Note that if you specify a range, the starting record is optional. The *filenum,* the *record,* the *start,* and the *end* are all names that you must supply.

LOCK 〚# 〛 *filenum* 〚,{ *record* ¦ 〚*start*〛 **TO** *end*} 〛

6.12   Reference Input/Output Notation

Program examples, user input, and program output are shown throughout this chapter. Typically, program lines are shown first, followed by descriptive text showing program output. In cases where there is user input, this input is shown in a different, darker type to distinguish it from the program listing or output from BASIC. Program examples listed either with line labels or without line numbers should be compiled with the /N switch.

The following example illustrates these conventions:

```
10 INPUT X
20 PRINT X "squared is" X^2
30 END
```

Output for this program, along with user input (231R), follows:

```
? 231
```

231   squared is 53361

ABS Function

■ **Syntax**

ABS(x)

■ **Action**

Returns the absolute value of the numeric expression x

■ **Example**

```
PRINT ABS(7*(-5))
```
Output:
```
35
```

■ **Syntax**

ASC(x$)

■ **Action**

Returns a numerical value that is the ASCII code for the first character of the string x$

■ **Remarks**

If x$ is null, an "Illegal function call" error is returned.

See the **CHR$** function for details on ASCII-to-string conversion.

■ **Example**

```
10 X$="TEST"
20 PRINT ASC(X$)
```

Output:

84

ATN Function

■ **Syntax**

ATN(x)

■ **Action**

The **ATN** function returns the arctangent of x, or the angle whose tangent is x

■ **Remarks**

The result is given in radians, and is in the range -2 to 2 radians, where $=$ 3.141593. The expression x may be any numeric type, but ATN is evaluated by default in single precision.

■ **Example**

```
10 X = 3
20 PRINT ATN(X)
```

Output:

```
1.249046
```

Thus an angle of 1.249046 radians has a tangent of 3.

■ **Syntax**

BEEP

■ **Action**

Sounds the speaker

■ **Remarks**

The **BEEP** statement sounds the ASCII bell character. This statement has the same effect as

PRINT CHR$(7)

in nongraphics versions of Microsoft BASIC.

■ **Example**

This example checks to see if X is out of range. If X is less than 20, the computer brings it to your attention by beeping:

20 IF X < 20 THEN BEEP

BLOAD Statement

■ **Syntax**

BLOAD *filespec* [*,offset*]

■ **Action**

Loads a specified memory image file into memory from any input device

■ **Remarks**

The device designation portion of the *filespec* is optional. The file name, not including the device designation, may be 1 to 8 characters long.

The *offset* is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address at which loading is to start in the segment declared by the last **DEF SEG** statement. The **BLOAD** statement allows a program or data that has been saved as a memory image file to be loaded anywhere in memory. A memory image file is a byte-for-byte copy of what was originally in memory.

If the offset is omitted, the segment address and offset contained in the file (i.e., the address specified by the **BSAVE** statement when the file was created) are used. Therefore, the file is loaded into the same location from which it was saved.

If offset is specified, the segment address used is the one given in the most recently executed **DEF SEG** statement. If no **DEF SEG** statement has been given, the BASIC data segment will be used as the default (because it is the default for **DEF SEG**).

Caution

   BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. The user must be careful not to load over BASIC or the operating system. When loading and saving graphic images, beware of problems associated with different memory configurations for different **SCREEN** modes.

■ **See Also**

BSAVE

■ **Example**

```
10 'Load subroutine at 60:F000
20 DEF SEG=&H6000    'Set segment to 6000 Hex
30 BLOAD "PROG1", &HF000    'Load PROG1
```

This example sets the segment address at &H6000 and loads PROG1 at &HF000.

BSAVE Statement

■ Syntax

BSAVE *filespec,offset,length*

■ Action

Transfers the contents of the specified area of memory to any output device

■ Remarks

The device designation portion of the *filespec* is optional. The file name, not including the device specification, must be 1 to 8 characters long.

The *offset* is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address to start saving from in the segment declared by the last **DEF SEG** statement.

The *length* is a numeric expression returning an unsigned integer in the range 1 to 65535. This is the length in bytes of the memory image file to be saved.

The *filespec, offset,* and *length* are all required in the syntax.

The **BSAVE** command allows data or programs to be saved as memory image files on disk. A memory image file is a byte-for-byte copy of what is in memory.

If the offset is omitted, a "Bad file name" error message is issued and the save is terminated. A **DEF SEG** statement must be executed before the **BSAVE**. The last known **DEF SEG** address will be used for the save.

If length is omitted, a "Bad file name" error message is issued and the save is terminated.

When loading and saving graphic images, beware of problems associated with different memory configurations for different **SCREEN** modes.

■ **See Also**

BLOAD

■ **Example**

```
10 'Save PROG1
20 DEF SEG=&H6000
30 BSAVE "PROG1", &HF000, 256
```

This example saves 256 bytes starting at 6000:F000 in the file PROG1.

CALL Statement

■ **Syntax**

CALL *name*[(*argumentlist*)]

■ **Action**

Calls an assembly language subroutine or a compiled routine written in another high-level language

■ **Remarks**

The *name* is a numeric variable. The value of this variable is the starting address in memory of the subroutine. The *name* may not be an array variable name.

The *argumentlist* is a comma-separated list of variable names that are passed to the external subroutine. The *argumentlist* may contain only variables.

The **CALL** statement is one way to transfer program flow to an external subroutine. (See also the **USR** function.)

■ **Example 1**

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
.
.
.
```

Line 110 starts the subroutine at memory address hex HD000. In line 120, the variables I, J, and K are passed as arguments to the subroutine.

■ **Compiler/Interpreter Differences**

In compiled programs, **CALL** can be used to invoke compiled BASIC sub-programs.

A BASIC compiler program does not require line 110 in the above example because the linker assigns the address of MYROUT at load time.

82

Additional differences for the compiler are:

1. The *name* is the name of the subroutine that is called. The name can be 1 to 31 characters long.

   If CALL invokes an assembly language subroutine, then the *name* must be recognized by Microsoft LINK as a global symbol. That is, *name* must be a PUBLIC symbol in an assembly language routine.

2. Since the GW-BASIC Compiler allows strings to be up to 32767 bytes long, the string descriptor requires four bytes rather than three as in the interpreter. The four bytes are: low byte, high byte of the length, followed by low byte, high byte of the address. If the assembly language routine uses string arguments, you may need to recode it to account for this difference.

3. For both interpreted and compiled programs, *argument-list* contains the variables or constants that **CALL** passes to the subroutine. In the compiler, variables or constants can also be passed to a compiled BASIC subroutine with the **SHARED** statement inside the subroutine, or with the **SHARED** attribute to **COMMON**, **DIM**, or **REDIM** in the main program.

■ **See Also**

COMMON, DIM, REDIM, SHARED

■ **Example 2**

This program, "combine.bas", calls three BASIC subprograms, "split", "strip", and "printout". The split subprogram breaks the command line input (command$) into two separate file names and stores them in the array, "file$()". Next, the strip subprogram strips leading blanks from the second file name. Finally, the printout subprogram writes out the contents of these two files. The DOS output redirection symbol (>) on the command line sends this output to a third file.

If two file names are not given on the command line, the result is the BASIC error "Illegal function call".

This program should be compiled with the /N switch to turn off the line-numbering constraint.

```
dim file$(2)
cmd$ = command$
call split(cmd$, file$())      'separate command line
call strip(file$(2))           'strip leading blanks
```

83

CALL Statement

```
call printout(file$())              'send both files
end

sub split(c$,f$(1)) static
    mark = instr(c$," ")            'find first blank
    f$(1) = left$(c$,mark - 1)      'everything up to first blank
    f$(2) = mid$(c$,mark + 1)       'everything after first blank
end sub

sub strip(f$) static
    first$ = left$(f$,1)
    while first$ = " "
        lng = len(f$)               'look for first non-blank charac
        f$ = right$(f$,lng - 1)     'in file$(2)
        first$ = left$(f$,1)
    wend
end sub

sub printout(f$(1)) static
    for file% = 1 to 2              'loop executes twice:
        open f$(file%) for input as #1      'first time for file$(1)
        while not eof(1)                    'second time for file$(2)
            line input #1, temp$    'read file
            print temp$             'write file to standard output
        wend
        close #1
    next
end sub
```

Sample command line:

```
combine    main.bas  sub.bas  >  prog.bas
```

After this executes, the contents of "main.bas" and "sub.bas" will be added together and stored in "prog.bas".

■ **Syntax**

CALLS *name*[[(*argumentlist*)]]

■ **Action**

Calls an assembly language subroutine

■ **Remarks**

The **CALLS** statement has the same syntax as **CALL**, and the same purpose, except that **CALLS** passes the segmented addresses of arguments in the *argumentlist*, while **CALL** passes unsegmented addresses.

With the interpreter only, **CALLS** uses the segment address defined by the most recently executed **DEF SEG** statement to locate the routine being called.

■ **See Also**

CALL

CDBL Function

■ Syntax

CDBL(*expression*)

■ Action

Converts the numeric expression *expression* to a double precision number

■ Example

```
10 LET PI = 22/7
20 PRINT PI,CDBL(PI)
```

Output:

```
3.142857      3.142857074737549
```

■ **Syntax**

CHAIN [MERGE]*filespec* [,[*linenum*] [,ALL] [,DELETE *range*]]]

■ **Action**

Calls a program and passes variables to it from the current program

■ **Remarks**

The *filespec* is a string expression containing a name that conforms to operating system naming conventions for disk files or GW-BASIC rules for device specifications.

The *linenum* is a line number (or an expression that evaluates to a line number) in the called program. It is the starting point for execution of the called program. If you omit it, execution begins at the first line of the called program. The RENUM command does not affect *linenum*.

With the ALL option, every variable in the current program is passed to the called program. If you omit the ALL option, the current program must contain a COMMON statement to list the variables that are passed. (See COMMON statement for information about COMMON.)

If you use the ALL option, but do not use *linenumber,* you need a comma to hold its place. For example,

CHAIN "NEXTPROG",, ALL

is correct;

CHAIN "NEXTPROG", ALL

is incorrect. In the latter case, GW-BASIC assumes that ALL is a variable name and evaluates it as a line number expression.

The MERGE option allows you to bring a subroutine into the GW-BASIC program as an overlay. In other words, you can merge the current program and the called program. If you want to merge the called program, it must be an ASCII file.

After you use an overlay, you will probably want to delete it so you can bring in a new overlay. To do this, use the DELETE option.

CHAIN Statement

The RENUM command does affect the line numbers in *range*.

■ **Examples**

CHAIN is used in different ways in the two examples below. In the first, the two string arrays are dimensioned, and then declared as common variables. When the program gets to line 90, it chains to the other program, which loads the strings in B$. At line 90 of PROG2, control chains back to the first program at line 100. The first program then continues execution from that line. You can observe this process through the descriptive text that prints as the programs execute.

■ **Example 1**

```
10   REM This program demonstrates chaining using
15   REM COMMON to pass variables.
20   REM Save this module on disk as "PROG1".
30   DIM A$(2),B$(2)
40   COMMON A$(),B$()
50   A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED"
60   A$(2)="VALUES BEFORE CHAINING."
70   B$(1)=""
80   B$(2) =""
90   CHAIN "PROG2"
100  PRINT
110  PRINT B$(1)
120  PRINT
130  PRINT B$(2)
140  PRINT
150  END
```

```
10   REM The statement "DIM A$(2),B$(2)"
15   REM may only be executed once;
20   REM hence, it does not appear in this module.
30   REM Save this module on the disk as "PROG2".
40   COMMON A$(),B$()
50   PRINT
60   PRINT A$(1);A$(2)
70   B$(1)="NOTE HOW THE OPTION OF SPECIFYiNG
A STARTING LINE NUMBER"
80   B$(2)="WHEN CHAINING AVOIDS THE DIMENSION
STATEMENT IN 'PROG1'."
90   CHAIN "PROG1",100
100  END
```

■ **Example 2**

The second example illustrates the **MERGE**, **ALL**, and **DELETE**
options. After the first program loads A$, control chains to line 1010 of the
second program. At the second program's line 1040, it chains to line 1010
of the third program, keeping all variables and deleting all the second
program's lines. Control passes to the third program. You can observe this
process through the descriptive text that prints as the programs execute.

```
10   REM This program demonstrates chaining using
15   REM the MERGE, ALL, and DELETE options.
20   REM Save this module on disk as "MAINPRG".
30   A$="MAINPRG"
40   CHAIN MERGE "OVRLAY1",1010,ALL
50   END


1000   REM Save this module on disk as "OVRLAY1".
1010   PRINT A$; " HAS CHAINED TO OVRLAY1."
1020   A$="OVRLAY1"
1030   B$="OVRLAY2".
1040   CHAIN MERGE "OVRLAY2",1010,ALL,DELETE 1000-1050
1050   END


1000   REM Save this module on the disk as "OVRLAY2"
1010   PRINT A$; " HAS CHAINED TO ";B$:"."
1020   END
```

Note

The **CHAIN** statement with **MERGE** option leaves the files open and
preserves the current **OPTIONBASE** setting.

If you omit the **MERGE** option, **CHAIN** does not preserve variable
types or user-defined functions for use by the chained program. That
is, you must restate in the chained program any **DEFINT**, **DEFSNG**,
DEFDBL, **DEFSTR**, or **DEFFN** statements containing shared vari-
ables.

When you use the **MERGE** option, you should place user-defined func-
tions before any **CHAIN MERGE** statements in the program. Other-
wise, the user-defined functions will be undefined after the merge is
complete.

CHAIN Statement

■ **Compiler/Interpreter Differences**

The Microsoft GW-BASIC Compiler does not support the **ALL, MERGE, DELETE**, and *linenumber* options to **CHAIN**. Thus, the statement syntax:

CHAIN *filespec*

For this reason, you should use **COMMON** to pass variables from one program to another. In addition, you should be careful when a chained-to program chains back to the chaining program; since you cannot specify a *linenumber* to return to, execution starts again at the beginning of the chaining program. There is a danger of going into an "endless loop" if this happens. (See the example for a way to avoid this problem.) The **CHAIN** statement leaves the files open during chaining.

The **BASCOM20.LIB** does not support the use of **CHAIN** with **COMMON**. Therefore, both the chaining and chained-to programs must use the default **BASRUN20.LIB**; that is, the chaining and chained-to programs must be compiled *without* the /O option.

In programs compiled with **BASRUN20.LIB**, files are left open during chaining.

See the "COMMON Statement" for more information about chaining with **COMMON**.

■ **See Also**

CALL, COMMON

■ **Example 3**

This example converts a number in any base to a decimal number. The base and the number are input in the main program, "main.bas", which then chains to a second program, "digit.bas", This second program splits the number from the main program into separate digits, converts those string values to numeric values, and stores the numeric values in an array, a(). Control then chains to a third program "dec.bas", which actually changes the number to a decimal number, deallocates (with **ERASE**) the array in which the digits were stored, and prints the decimal number. Control then chains back to the main program, which repeats the process, as long as the value input for the base b is not zero. Note the use of **COM-MON** in all three programs to share variables.

Compile this program with the /N switch to turn off the line-numbering constraint.

```
rem ** This program is main.bas **
common a(1),n$,b,ln
input "base,number:   ",b,n$
print
while b
   ln = len(n$)
   dim a(ln)
   chain "digit"
wend
end

rem ** This program is digit.bas **
common a(1),n$,b,ln
m = ln -1
for j = 0 to m
   a$ = mid$(n$,j+1,1)
   if a$ < "A" then a(m-j) = val(a$)
   else  a(m-j) = asc(a$) - 55
next
chain "dec"

rem ** This program is dec.bas **
common a(1),n$,b,ln
dec = 0
for i = 0 to (ln-1)
   dec = dec + a(i)*b^i
next
erase a
print "Decimal # = ";dec : print
print "Input 0 for base to end program."
chain "main"
```

Sample output:

```
base,number:   16,43E

Decimal # =  1086

Input 0 for base to end program.
base,number:  0,
```

CHDIR Statement

■ **Syntax**

CHDIR "$[d:]pathname$"

■ **Action**

Changes the current operating directory

■ **Remarks**

The argument $d:$ is an optional drive specification.

The string *pathname* specifies the name of the directory which is to be the current directory. **CHDIR** works exactly like the DOS command **CHDIR**. The *pathname* must be a string of less than 128 characters.

■ **See Also**

MKDIR, RMDIR

■ **Example 1**

This makes SALES the current directory:

```
CHDIR "SALES"
```

■ **Example 2**

This changes the current directory to USERS on drive B. It does *not*, however, change the default drive to B.

```
CHDIR "B:USERS"
```

■ **Syntax**

CHR$ *(code)*

■ **Action**

Returns a one-character string whose ASCII code is *code*

■ **Remarks**

CHR$ is commonly used to send a special character to the screen or printer. For instance, you could send the BELL character (CHR$ (7)) as a preface to an error message, or you could send a form feed (CHR$ (12)) to clear a terminal screen and return the cursor to the home position.

■ **Example**

```
PRINT CHR$(66)
```

Output:

```
B
```

See the "ASC Function" for details on ASCII-to-numeric conversion.

CINT Function

■ Syntax

CINT(*expression*)

■ Action

Converts the numeric expression *expression* to an integer by rounding the fractional part of the expression.

■ Remarks

If *expression* is not in the range -32768 to 32767, an "Overflow" error message is generated.

CINT differs from the FIX and INT functions, which return integers by truncating the fractional part.

See "CDBL" and "CSNG" for details on converting numbers to the double precision and single precision data type, respectively.

■ See Also

CDBL, CSNG, FIX, INT

■ Example

```
PRINT CINT(45.67), CINT(-45.67)
```
Output:

```
46          -46
```

■ **Syntax**

CIRCLE [STEP] (x,y),radius[,color[,start,end[,aspect]]]

■ **Action**

Draws an ellipse or circle with the specified center and radius

■ **Remarks**

The **STEP** option makes the specified x and y coordinates relative to the "most recent point," instead of absolute, mapped coordinates.

The other arguments to **CIRCLE** are the following:

x,y	The x and y coordinates for the center of the ellipse
radius	The radius of the ellipse in the current coordinate system
color	The numeric symbol for the color desired (see the "COLOR Statement.") The default color is the foreground color.
start	The start and end angles in radians. They may range from -2π to 2π, where $\pi = 3.141593$. These angles allow you to specify where an ellipse will begin and end. If the start or end angle is negative, then **CIRCLE** draws a straight line from that point on the ellipse to the center point, and treats the angle as if it were positive. Note that this is not the same as adding 2π to the negative angle. The start angle may be less than the end angle.
aspect	The aspect ratio, or the ratio of the y radius to the x radius. When you do not specify *aspect*, **CIRCLE** draws a round circle. The default value for aspect depends on your hardware display configuration.
	If the aspect ratio is less than one, the radius is the x radius. If aspect is greater than one, *radius* is equal to the y radius.

The last point that **CIRCLE** references after drawing an ellipse is the center of the ellipse.

It is not an error to supply coordinates that are outside the screen or viewport.

CIRCLE Statement

You can show coordinates as absolutes, as in the syntax shown above, or you can use the **STEP** option to show the position of the center point in relation to the previous point of reference. The syntax of the **STEP** option is:

STEP (*xoffset, yoffset*)

For example, if the previous point of reference is (10,10), then

STEP (10,5)

refers to a point offset 10 from the current *x* coordinate and offset 5 from the current *y* coordinate, that is, the point (20,15).

■ **Example 1**

Assume that the last point plotted was 100,50. Then,

```
SCREEN 2
CIRCLE (200,200),50
```

and

```
SCREEN 2
CIRCLE STEP (100,150),50
```

will both draw a circle with center at 200,200 and radius 50. The first statement uses absolute notation; the second uses relative notation.

■ **Example 2**

The following draws a circle on the screen, with the top left quarter removed.

```
SCREEN 2
PI = 3.141593
CIRCLE (100,150), 50,, -PI, -PI/2
```

■ **Example 3**

Since the aspect ratio is less than one, this example draws an ellipse that has a wide horizontal axis and a narrow vertical axis. This also means that 60 is the *x* radius, so the length of the horizontal axis equals 120, or 2 times 60:

```
SCREEN 2
CIRCLE (160,100), 60,,,, 5/18
```

96

■ **Syntax**

CLEAR [[,[[*location*]] [[,*stack*]]

■ **Action**

The **CLEAR** statement performs the following actions:

• Closes all files.

• Clears all **COMMON** variables.

• Resets numeric variables and arrays to zero.

• Resets the stack and string space.

• Resets all string variables to null.

• Releases all disk buffers.

• Resets all **DEF FN** and **DEF** *type* statements.

■ **Remarks**

The *location* is the memory location of the highest location available for use by BASIC. The *stack* parameter sets aside the given number of byes of stack space.

■ **Examples**

The following statement sets all numeric variables to zero, all string variables to null, and closes all files opened by BASIC:

CLEAR

The following statement clears the data and sets the stack to 2,000 bytes:

CLEAR , 2000

■ **Compiler/Interpreter Differences**

This statement requires modification of interpreted BASIC programs when they are used with the GW-BASIC Compiler. In interpreted programs, **CLEAR** resets all **DEF FN** and **DEF** *type* statements; the compiler, on the other hand, does not reset these statements. These declarations are fixed at compile time and may not vary.

CLOSE statement

■ **Syntax**

CLOSE[[#] *filenumber* [,[#] *filenumber*...]]

■ **Action**

Concludes I/O to a file

■ **Remarks**

The **CLOSE** statement is complementary to the **OPEN** statement.

The *filenumber* is the number under which the file was opened. A **CLOSE** with no arguments closes all open files.

The association of a particular file and a file number terminates upon execution of a **CLOSE** statement. You may then reopen the file using the same or a different file number. Once you close a file, you may use that file's number for any unopened file.

A **CLOSE** for a sequential output file writes the final buffer of output.

The **SYSTEM, CLEAR,** and **END** statements and the **NEW** and **RESET** commands always close all files automatically.

■ **Example**

```
CLOSE #1,#2
```

■ **Syntax**

CLS [0¦1¦2]

■ **Action**

Clears the screen.

■ **Remarks**

There are several ways to use **CLS**:

Statement	Effect
CLS 0	Clears the screen of all text and graphics.
CLS 1	Clears only the graphics viewport.
CLS 2	Clears only the text window.
CLS	If the graphics viewport is active, then **CLS** with no argument clears only the viewport. If the graphics viewport is inactive, then **CLS** clears the text window.

■ **Example**

```
10   CLS
```

COLOR statement

■ Syntax

COLOR [[foreground]][,[background]][,border]]]	'Screen 0
COLOR [background][,[palette]]]	'Screen 1
COLOR [[foreground]][,[background]]]]	'Screens 7-10

■ Action

Selects display colors

■ Remarks

In general, COLOR allows you to select the foreground and background colors for the display. In SCREEN 1 no foreground color can be selected, but one of two four-color palettes can be selected for use with graphics statements. In SCREEN 0 a border color also can be selected. The different syntaxes and effects that apply to the various screen modes are described below:

Mode	Effect
0	Modifies the current default text foreground and background colors, and the screen border. The *foreground* color must be an integer expression in the range 0-31. It is used to determine the "foreground" color in text mode, which is the default color of text. Sixteen colors can be selected with the integers 0-15. A blinking version of each color can be selected by adding 16 to the color number; for example blinking color 7 is equal to 7 + 16 or 23. Thus the legal integer range for *foreground* is 0-31.
	The *background* color must be an integer expresssion in the range 0-7, and is the color of the background for each text character. Blinking colors are not permitted.
	The *border* color is an integer expression in the range 0-15 and is the color used when drawing the screen border. Blinking colors are not permitted.
	If no arguments are provided to COLOR then the default color for *background* and *border* is black (color 0), and for *foreground*, is as described in the "SCREEN Statement."
1	In mode 1, the COLOR statement has a unique syntax that includes a *palette* argument that is an odd or even integer expression. This argument determines which set of display

colors to use when displaying particular color numbers.

For non-EGA hardward configurations, the default color settings for the *palette* parameter are equivalent to the following:

```
COLOR ,0      'Same as the next three PALETTE statements
              '1 = green, 2 = red, 3 = yellow

COLOR ,1      'Same as the next three PALETTE statements
              '1 = cyan, 2 = magenta, 3 = hi. intens. white
```

With the EGA, the default color settings for the *palette* parameter are equivalent to the following:

```
COLOR ,0      'Same as the next three PALETTE statements
PALETTE 1,2   'Attribute 1 = color 3 (green)
PALETTE 2,4   'Attribute 2 = color 5 (red)
PALETTE 3,6   'Attribute 3 = color 6 (brown)

COLOR ,1      'Same as the next three PALETTE statements
PALETTE 1,3   'Attribute 1 = color 3 (cyan)
PALETTE 2,5   'Attribute 2 = color 5 (magenta)
PALETTE 3,7   'Attribute 3 = color 15 (white)
```

Note that a **COLOR** statement will override previous **PALETTE** statements.

2   No effect. An "Illegal Function Call" message results if **COLOR** is used in this mode.

7-10 In these modes, no *border* color can be specified. The graphics background is given by the *background* color number, which must be in the valid range of color numbers appropriate to the screen mode. See the "SCREEN Statement" for more details. The *foreground* color argument is the default line drawing color.

Arguments outside valid numeric ranges result in "Illegal function call" errors.

The foreground color may be the same as the background color, making displayed characters invisible. The default background color is black, or color number 0, for all display hardware configurations and all screen modes.

With the Enhanced Graphics Adapter (EGA) installed, the **PALETTE** statement gives you flexibility in assigning different display colors to the actual color number ranges for the *foreground, background* and *border* colors discussed above. See the "PALETTE Statement" for more details.

COLOR statement

■ **Examples**

The following series of examples show **COLOR** statements and their effects in the various screen modes.

```
SCREEN 0
COLOR  1, 2, 3  'foreground=1, background=2, border=3

SCREEN 1
COLOR  1,0   'foreground=1, even palette number
COLOR  2,1   'foreground=2, odd palette number

SCREEN 7
COLOR  3,5   'foreground=3, background=5

SCREEN 8
COLOR  6,7   'foreground=6, background=7

SCREEN 9
COLOR  1,2   'foreground=1, background=2
```

102

■ Syntax

COM(n) ON
COM(n) OFF
COM(n) STOP

■ Action

Enables or disables event trapping of communications activity on the specified port

■ Remarks

The parameter n is the number of the communications port. The range for n is specified by the implementor.

The **COM ON** statement enables communications event trapping by an **ON COM** statement. If you specify a non-zero line number in the **ON COM** statement while trapping is enabled, GW-BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, it executes the **ON COM** statement.

COM OFF disables communications event trapping. If an event takes place, it is not remembered.

COM STOP disables communications event trapping, but if an event occurs, it is remembered. If there is a subsequent **COM ON** statement, the remembered event will be successfully trapped.

■ Example

This enables error trapping of communications activity on channel 1:

```
10 COM(1) ON
```

■ Compiler/Interpreter Differences

See "ON COM Statement."

COMMAND$ Function

■ Function Syntax

COMMAND$

■ Action

Returns the command line used to invoke the program

■ Remarks

This function returns the complete command line, including any optional parameters, after first stripping all leading blanks from the command line, and converting all letters to upper case (capital letters).

■ Compiler/Interpreter Differences

COMMAND$ works only in the compiler.

■ Example

The following program computes the logarithm to the base 10 of any number greater than zero that is entered after "log" (the name of the executable file) on the command line:

```
bs = 10
power = 1
sign = 1
lg = 0
x = val(command$)
test1: if x > 0 then goto test2 _
    else _
        print "log(";command$;") not defined."
        print "Input must be greater than zero."
        end
test2: if x >= 1 then goto test3 _
    else _
        x = 1/x
        sign = -1
test3: if x < 100 then goto main _
    else _
        while x >= 100
            x = x/10
            lg = lg + 1
        wend
```

```
main:
    while abs(bs - 1) > .0000001
        if bs > x then goto newval _
        else _
            x = x/bs
            lg = lg + power
        newval:
            bs = sqr(bs)
            power = power/2
    wend
    print "log(";command$;") = ";lg*sign
```

Command line and output:

log 6.78

```
log 6.78 =  .8312299
```

Command line and output:

log -1

```
log(-1) not defined.
Input must be greater than zero.
```

COMMON Statement

■ **Syntax**

COMMON *variablelist*

■ **Action**

Passes variables to a chained program

■ **Remarks**

The **COMMON** statement is used in conjunction with the **CHAIN** state-
ment. **COMMON** statements may appear anywhere in a program, though
it is recommended that they appear at the beginning. The same variable
cannot appear in more than one **COMMON** statement. Array variables
are specified by appending parentheses () to the variable name. If all vari-
ables are to be passed, use **CHAIN** with the **ALL** option and omit the
COMMON statement.

Some Microsoft products allow the number of dimensions in the array to be
included in the **COMMON** statement. GW-BASIC will accept that syn-
tax, but will ignore the numeric expression itself. For example, the follow-
ing statements are both valid and are considered equivalent:

```
COMMON A()
COMMON A(3)
```

The number in parentheses is the number of dimensions, not the dimen-
sions themselves. For example, the variable A(3) in this example might
correspond to the following **DIM** statement:

```
DIM A(5,8,4).
```

■ **Example**

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
  .
  .
```

■ **Compiler/Interpreter Differences**

The following is the syntax for **COMMON** in the compiler:

COMMON [SHARED] /*blockname*/ *variablelist*

In the compiler, **COMMON** can be used to pass variables to a compiled BASIC subprogram. (See "SUB...END SUB.")

With the BASIC interpreter, you may put **COMMON** statements anywhere in a program. With the compiler, however, the **COMMON** statement must appear in a program before any executable statements. All statements are executable, except the following:

COMMON
DEF *type*
DIM (for static arrays)
OPTION BASE
REM
$ *name* (all compiler metacommands)

If you use static array variables in a **COMMON** statement, then you must declare them in a preceding **DIM** statement. If an array is dynamic, the **DIM** statement follows the **COMMON** statement, since a **DIM** statement is an executable statement when the array it dimensions is dynamic.

SHARED is an optional attribute. Use **SHARED** to pass variables from a main program to a BASIC subprogram within the same module. This way, you don't need the **SHARED** statement, or another **COMMON** statement, within the subprogram. You also need the **SHARED** attribute with **COMMON** when the subprogram being called is in a separately compiled module.

The *blockname* is any valid BASIC identifier up to 31 characters long. Use *blockname* when not every subprogram shares all variables in the *variablelist*. Items in a named **COMMON** statement are not preserved across a chain to a new program.

The *variablelist* is a list of variables and arrays that are used by subprograms or chained-to programs. The same variable cannot appear in more than one COMMON statement. *Static* array variables are specified by appending parentheses () to the variable name. *Dynamic* array variables are specified by appending to the variable name, where *num* is an integer constant indicating the number of dimensions in the array.

COMMON Statement

When you use **COMMON** with **CHAIN**, you must use the **BASRUN20.EXE** module (this means you should compile your program without the /O option). Also, both the chaining program and the chained-to program require a **COMMON** statement. With the GW-BASIC compiler, the *order* of variables must be the same for all **COMMON** statements communicating between chaining and chained-to programs. If the *size* of the common region in the chained-to program is smaller than the region in the chaining program, the extra **COMMON** variables in the chaining program are ignored. If the size of the common region in the chained-to program is larger, the additional **COMMON** variables are initialized to zeros and null strings.

To ensure that programs can share common areas, place **COMMON** declarations in a single "include" file and use the $**INCLUDE** statement in each program.

■ **See Also**

CHAIN, SHARED, SUB...END SUB

■ **Example**

This program fragment shows the use of an include file to share **COMMON** statements among programs:

```
rem ** This file is menu.bas **
rem $include:'comdef.bas'
.
.
.
chain "prog1"
end

rem ** This file is prog1.bas **
rem $include:'comdef.bas'
.
.
.
end

rem ** This file is comdef.bas **
dim A(100),B$(200)
common I,J,K,A()
common A$,B$(),X,Y,Z
rem ** End comdef.bas **
```

■ **Syntax**

CONT

■ **Action**

Continues program execution after a BREAK has been typed or a **STOP** statement has been executed.

■ **Remarks**

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an **INPUT** statement, execution continues with the reprinting of the prompt string, which is a question mark (?) by default.

CONT is usually used in conjunction with **STOP** for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with **CONT** or a direct mode **GOTO**, which resumes execution at a specified line number. **CONT** may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

■ **Example**

See **STOP** example.

■ **Compiler Differences**

The **CONT** command is not supported by the compiler.

COS Function

■ **Syntax**

COS(x)

■ **Action**

Returns the cosine of x, where x is in radians

■ **Remarks**

The calculation of **COS** is performed in single precision, unless you specify the **/D** option when invoking BASIC. **COS** is then performed in double precision if the variable that receives the value of the cosine is a double precision variable or if you have made x a double precision number with the number sign (#).

You can convert an angle measurement from degrees to radians by multiplying the degrees by $\pi/180$, where $\pi = 3.141593$.

■ **Example 1**

```
10 X=2*COS(.4)
20 PRINT X
```
Output:
```
1.842122
```

■ **Example 2**

This prints the cosine of 60 degrees:
```
100 PI = 3.141593
110 DEGREES = 60
120 RADIANS = DEGREES*PI/180
130 PRINT COS(RADIANS)
```
Output:
```
        .5
```

■ **Syntax**

CSNG(x)

■ **Action**

Converts x to a single precision number

■ **Example**

```
10 A# = 975.3421115#
20 PRINT A#   CSNG(A#)
```

This prints the following values:

```
 975.3421115    975.3421
```

See the "CINT Function" and the "CDBL Function" for information on converting numbers to the integer and double precision data types, respectively.

CSRLIN Function

■ **Syntax**

CSRLIN

■ **Action**

Obtains the current line position of the cursor in a numeric variable

■ **Remarks**

To return the current column position, use the **POS** function.

■ **Example**

```
10 Y = CSRLIN        'Record current line.
20 X = POS(0)        'Record current column.
30 LOCATE 24,1
40 PRINT "HELLO"
50 LOCATE X,Y        'Restore position to old line and column
```

■ **Syntax**

CVI(*2bytestring*)
CVS(*4bytestring*)
CVD(*8bytestring*)

■ **Action**

Convert string values to numeric values.

■ **Remarks**

Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts *2bytestring* to an integer. CVS converts a *4bytestring* to a single precision number. CVD converts an *8bytestring* to a double precision number.

The CVI, CVS, and CVD functions do not change the bytes of the actual data. They change only the way BASIC interprets those bytes.

■ **Example**

```
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
```

■ **See Also**

MKI$, MKS$, MKD$

DATA Statement

■ **Syntax**

DATA *constant1* [[,*constant2*]]...

■ **Action**

Store the numeric and string constants that are used by the program's
READ statement(s).

■ **Remarks**

DATA statements are nonexecutable. You may place them anywhere in
the program.

A **DATA** statement may contain as many constants as will fit on a line.
These constants should be separated by commas.

You may use any number of **DATA** statements in a program. **READ**
statements read **DATA** statements in order (by line number). You can
think of the items in several **DATA** statements as one continuous list of
items, regardless of how many items are in a statement or where the state-
ment is placed in the program.

The constants in a **DATA** statement may be in any format; i.e., fixed-
point, floating-point, or integer. (No numeric *expressions* are allowed in the
list.)

You do not need to put double quotation marks around string constants in
DATA statements, unless they contain commas, colons, or significant lead-
ing or trailing spaces.

The variable type (numeric or string) that you give in the **READ** state-
ment must agree with the corresponding constant in the **DATA** statement.

A **READ** statement can reread the data in one or more of your program's
DATA statements if you use the **RESTORE** statement.

114

■ **Example**

See the "READ Statement" for an example.

DATE$ Function

■ Syntax

DATE$

■ Action

Retrieves the current date or sets the date, using the DATE$ statement

■ Remarks

The DATE$ function returns a ten-character string in the form *mm-dd-yyyy,* where *mm* is the month (01 through 12), *dd* is the day (01 through 31), and *yyyy* is the year (1980 through 2099).

■ Example

```
10  DATE$ = "5/20/85"
20  PRINT DATE$
```

This prints the following:

```
05-20-1985
```

Note that the DATE$ function prints a zero in front of the month (05) to make it have two digits. Also, it prints the answer with the month, day and year separated by dashes (-), even though they were entered in the DATE$ statement with slashes (/).

■ **Syntax**

DATE$ =*stringexpression*

■ **Action**

Sets the current date

■ **Remarks**

This statement complements the **DATE$** function, which retrieves the current date. The *stringexpression* must be a string in one of the following forms:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

■ **Example**

```
10  DATE$="07-01-1983"
```

The current date is set to July 1, 1983.

DEF FN Statement

■ Syntax

DEF FN*name*[(*parameterlist*)]= *functiondefinition*

■ Action

Defines and names a function

■ Remarks

The *name* must be a legal variable name. This name, preceded by **FN**, becomes the name of the function.

The *parameterlist* is a list of variable names, separated by commas. When the function is called, it replaces these variables on a one-to-one basis with the values the program supplies.

The *functiondefinition* is an expression that performs the operation of the function. It is limited to one logical line. Variable names that appear in this expression are local to the expression and serve to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

Argument variables or values that appear in the function call replace the variables in the parameter list on a one-to-one basis.

DEF FN can define either numeric or string functions. **DEF FN** returns a string value if *name* is a string variable name, and a numeric value if *name* is a numeric variable name. If the type (string or numeric) of *functiondefinition* or *expression* does not match the type of **DEF FN***name*, then a "Type mismatch" error occurs.

If the function is numeric, **DEF FN***name* gives the value it returns to the calling statement the precision specified by *name*. For example, if *name* specifies a double precision variable, then the value of **DEF FN***name* is in double precision, regardless of the precision of *functiondefinition* or *expression*.

Warning

> Your program must define a function with a **DEF FN** statement before it can call it. If your program calls a function before it is defined, an "Undefined user function" error occurs.
>
> User-defined functions cannot appear inside other multiline functions, nor can they appear inside **IF...THEN...ELSE, FOR...NEXT,** or **WHILE...WEND** blocks.
>
> Your program cannot contain recursive function definitions; that is, a function cannot be defined in terms of itself.

■ **Example 1**

In this example, line 20 defines the function "fnarea", which calculates the area of a circle with radius "r." Line 40 calls the function.

```
10 pi = 3.141593
20 def fnarea(r) = pi * r^2
30 input "Radius";radius
40 print "Circle area is ";fnarea(radius)
```

Output:

```
Radius? 2
Circle area is 12.56637
```

■ **Compiler/Interpreter Differences**

The compiler supports multi-line function definitions, as well as single-line function definitions. The syntax of a multiline **DEF FN** is as follows:

DEF FN*name*[*(parameterlist)*]
.
.
.
FN*name* = *expression*
.
.
.
END DEF

DEF FN Statement

The *expression* defines the value returned by the function. A multiline function ends with an **END DEF** statement or the optional **EXIT DEF** statement.

Note

The **RETURN** statement is *not* equivalent to **END DEF** or **EXIT DEF**. Using a **RETURN** statement to exit a multiline function will cause a severe unrecoverable error.

■ **Example**

The following example contains a function definition that converts an angle measure in degrees, minutes and seconds to an angle measure in radians. (An angle must be given in radians for the trigonometric functions of BASIC to return a meaningful answer.)

```
def fndegrad(d,m,s)
   pi = 3.14159263
   d = d + m/60 + s/3600
   fndegrad = d * (pi/180)
end def
deg = 45 : min = 10
print tab(5);"Angle measurement";tab(38);"SINE"
print
for sec = 10 to 50 step 10
   print tab(5);deg;chr$(248);",";min;"'",";sec;chr$(34);
   rad = fndegrad(deg,min,sec)
   print tab(35);sin(rad)
next
end
```

Output:

```
     Angle measurement              SINE

       45° 10' 10"                 .7091949
       45° 10' 20"                 .7092291
       45° 10' 30"                 .7092632
       45° 10' 40"                 .7092974
       45°,10',50"                 .7093316
```

■ **Syntax**

DEF SEG [[=*address*]]

■ **Action**

Assigns the current segment address to be referenced by a subsequent
BLOAD, BSAVE, CALL, CALLS, or **POKE** statement or by a **USR** or
PEEK function

■ **Remarks**

The *address* is a numeric expression returning an unsigned integer in the
range 0 to 65535.

The address specified is saved for use as the segment required by **BLOAD,
BSAVE, CALL, CALLS, POKE, USR,** and **PEEK.**

Entry of any value outside the *address* range of 0 through 65535 results in
an "Illegal function call" error. The previous value is retained when this
happens.

If you omit the *address* option, the GW-BASIC data segment is used by
default.

You must separate **DEF** and **SEG** with a space. Otherwise, GW-BASIC
interprets

DEF SEG=100

to mean "assign the value 100 to the variable DEFSEG."

■ **Example**

```
10 DEF SEG=&HB800 'Seg segment at B800 Hex
20 DEF SEG 'Restore segment to GW-BASIC data segment
```

121

DEF SEG Statement

▪ Compiler/Interpreter Differences

With the compiler, only the **BLOAD**, **BSAVE**, and **POKE** statements and the **PEEK** and **USR** functions reference **DEF SEG**.

■ **Syntax**

DEFINT *letter-range*
DEFSNG *letter-range*
DEFDBL *letter-range*
DEFSTR *letter-range*

■ **Action**

Declares variables of integer, single precision, double precision, or string type

■ **Remarks**

Any variable names beginning with the letters specified in *letter-range* are the type of variable specified by the last three letters of the statement, that is, either **INT** (integer), **SNG** (single precision), **DBL** (double precision), or **STR** (string). For example, the following statement declares all variables beginning with the letter A as string variables:

```
DEFSTR A
```

In the next example, all variables beginning with the letters K, L, M, X, Y or Z are designated integer variables:

```
DEFINT K-M,X-Z
```

A type declaration character such as %, !, #, or $ always takes precedence over a **DEF***type* statement.

Note

I!, I#, I$, and I% are all separate and distinct variables — each one can hold a different value.

DEF *type* Statements

■ Compiler/Interpreter Differences

The interpreter and the compiler process **DEF** *type* statements somewhat differently. The interpreter must scan a statement each time before it executes it. If the statement contains a variable that does not have an explicit type (signified by !, #, $, or %), the interpreter determines the current default type and uses it. In the example below, when the interpreter encounters line 20, it determines that the current default type for variables beginning with "I" is integer. Line 30 then changes this default type to single precision and loops back to line 20. The interpreter must rescan line 20 in order to execute it and this time IFLAG becomes a single precision variable:

```
10   DEFINT I
20   PRINT IFLAG
30   DEFSNG I
40   GOTO 20
```

In contrast, the compiler scans the text only once. Therefore, once a variable occurs in a program line, its type cannot be changed. The compiler, unlike the interpreter, does.not allow you to circumvent a **DEF** *type* statement by directing program flow around it.

You can see these differences in the output from the example program.

■ Example

The following program gives different results when you run it with the interpreter and with the compiler. The interpreter assigns variable types each time it scans a statement during program execution, so it allows the program to redeclare the variable type inside the **FOR...NEXT** loop. On the other hand, the compiler statically scans **DEF** *type* statements, assigning variable types at compile time, so line 160 applies only to occurrences of "T" variables in program lines after line 160.

```
100   TEST% = 1           ' Integer type
110   TEST! = 10          ' Single precision type
120   DEFINT T
130   FOR I = 1 TO 3
140       PRINT TEST
150       TEST = TEST + 20
160       DEFSNG T
170   NEXT
180   PRINT
190   TEST = TEST + 100
200   PRINT "TEST = ";TEST
210   PRINT "TEST% =";TEST%
220   PRINT "TEST! =";TEST!
```

```
Interpreter output:      Compiler output:
 1                            1
 10                          21
 30                          41

test =  150                  test =  110
test% = 21                   test% = 61
test! = 150                  test! = 110
```

DEF USR Statement

■ Syntax

DEF USR[[*digit*]] = *integer-expression*

■ Action

Specifies the starting address of an assembly language subroutine

■ Remarks

The *digit* may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If *digit* is omitted, **DEF USR0** is assumed. The value of *integer-expression* is the starting address of the USR routine.

Any number of **DEF USR** statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

■ Example

```
.
.
.
200 DEF  USR0=24000
210 X=USR0(Y^2/2.89)
.
.
.
```

■ **Syntax**

DELETE *linenumber[-]*
DELETE *[-]linenumber*
DELETE *linenumber1-linenumber2*

■ **Action**

Deletes program lines

■ **Remarks**

The **DELETE** command deletes lines from a program in the range supplied by a range of line numbers. In the first syntax above, all lines beginning with *linenumber* are deleted. In the second syntax, all lines up to and including *linenumber* are deleted. Finally, in the third syntax, all lines in the range *linenumber1-linenumber2* are deleted. BASIC always returns to command level after **DELETE** is executed. If *linenumber* does not exist, an "Illegal function call" error occurs.

■ **Examples**

Deletes line 40:

DELETE 40

Deletes lines 40 through 100, inclusive:

DELETE 40-100

Deletes all lines up to and including line 40:

DELETE -40

Deletes lines 40 through the end, inclusive:

DELETE 40-

DIM Statement

■ **Syntax**

DIM *variable(subscripts)* [, *variable(subscripts)*]...

■ **Action**

Specify the maximum values for array variable subscripts and allocate storage accordingly.

■ **Remarks**

If you use an array in your program without including the array in a **DIM** statement, the maximum value of the array's subscript(s) is assumed to be 10. If you use a subscript that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is 0, unless you specify otherwise with an **OPTION BASE** statement.

The **DIM** statement sets all the elements of the specified numerical arrays to an initial value of zero and all the elements of string arrays to null strings.

Theoretically, the maximum number of dimensions allowed in a **DIM** statement is 255. In reality, however, that number is impossible, since the name and punctuation also count as spaces on the line, and the line itself has a limit of 255 characters.

If you try to dimension an array variable with a **DIM** statement, but the array dimension has already been assigned the default value of 10, an "Array already dimensioned" error results. Therefore, it is good programming practice to put the required **DIM** statements at the beginning of a program, outside of any processing loops.

■ **Example**

```
10 DIM A(20)
20 FOR I=0 TO 20
30    READ A(I)
40 NEXT I
   .
   .
   .
```

■ **Compiler/Interpreter Differences**

The compiler scans the **DIM** statement instead of executing it. That is, **DIM** takes effect when it is encountered at compile time and remains in effect until the end of the program. It cannot be reexecuted at runtime.

The exception to the above is when the array is a dynamic array. (See the "$DYNAMIC Metacommand.") In this case, a **DIM** statement is considered an executable statement.

DRAW Statement

■ Syntax

DRAW "*subcommand-string*"

■ Action

Draws an object defined by *subcommand-string*

■ Remarks

The **DRAW** statement combines many of the capabilities of the other graphics statements into a graphics macro language, as described below under *Prefixes, Cursor Movement,* and *Other Commands.* The Graphics Macro Language defines a set of characteristics that comprehensively describe a particular image. In this case, the characteristics include motion (up, down, left, right), color, angle rotation, and scale factor.

Each of the following subcommands initiates movement from the current graphics position, this usually being the coordinate of the last graphics point plotted with another Graphics Macro Language command. The current position defaults to the center of the screen when a program is run.

■ Prefixes

The following prefix commands may precede any of the movement commands:

B   Move but don't plot any points.

N   Move but return to original position when done.

■ Cursor Movement

The following commands specify movement in units. The default unit size is one point; this unit size can be modified by the **S** command. If no argument is supplied, the cursor is moved one unit.

U $[\![n]\!]$ Move up scale factor n points

D $[\![n]\!]$ Move down

L $[\![n]\!]$ Move left

R $[\![n]\!]$ Move right

E $[\![n]\!]$ Move diagonally up and right

F $[\![n]\!]$ Move diagonally down and right

G $[\![n]\!]$ Move diagonally down and left

H $[\![n]\!]$ Move diagonally up and left

■ Other Commands

M x,y

Move absolute or relative. If x is preceded by a plus (+) or minus (−), the movement is relative to the current point; that is, x and y are added to the coordinates of the current graphics position and connected with that position by a line.

If no sign precedes x, the movement is absolute; that is, a line is drawn from the current cursor position to the point with coordinates x,y.

A n

Set angle n. The value of n may range from 0 to 3, where 0 is 0°, 1 is 90°, 2 is 180°, and 3 is 270°. Figures rotated 90° or 270° are scaled so they will appear the same size as with 0° or 180° on a monitor screen with the standard aspect ratio of 4/3.

TA n

Turn an angle of n degrees; n must be in the range −360° to 360°. If n is positive, rotation is counter-clockwise; if n is negative, rotation is clockwise. The following example draws spokes:

```
FOR D=0 TO 360   STEP 10
    DRAW "TA="+VARPTR$ (D) +"NU50"
NEXT D
```

C n

Set color n. The range of values that n can assume depends on the screen mode set with the **SCREEN** statement.

S n

Set scale factor n, which may range from 1 to 255. The scale factor multiplied by the distances given with **U**, **D**, **L**, **R**, or relative **M** commands gives the actual distance traveled.

X *string-expression*

Execute substring. This powerful command allows you to execute a second substring from a string. You can have one string expression execute another, which executes a third, and so on.

Numeric arguments can be constants like 123 or variable names.

DRAW Statement

P *paintcolor, bordercolor*

> The *paintcolor* is the paint attribute for a figure's interior, while *bordercolor* is the paint attribute for the figure's border. The color selected by *paintcolor* and the *bordercolor* depends on the palette chosen in the **COLOR** statement. "Tile" painting is not supported in **DRAW**.

■ Example 1

The following program draws a triangle in magenta and paints the interior cyan:

```
SCREEN 1
DRAW "C2"                  'Set color to magenta
DRAW "F60 L120 E60"        'Draw a triangle
DRAW "BD30"                'Move down into the triangle
DRAW "P1,2"                'Paint interior
INPUT "Press return to end",BYE$
```

■ Compiler/Interpreter Differences

This statement requires modification of interpreted BASIC programs when used with the compiler. Specifically, the compiler requires the **VARPTR$** form for variables. Statements such as

```
DRAW "XA$"
```

and

```
DRAW "TA = ANGLE"
```

(where A$ and ANGLE are variables) should be changed to

```
DRAW "X"+ VARPTR$ (A$)
```

and

```
DRAW "TA =" + VARPTR$ (ANGLE)
```

in the compiler.

The compiler does not support the

"X string-expression"

command. However, you can execute a substring by appending the character form of the address to X. For example, the following two statements are equivalent: the first works with the interpreter, the second with the compiler.

```
DRAW "XA$"
DRAW "X" + VARPTR$ (A$)
```

EDIT Command

■ **Syntax**

EDIT *linenumber*

■ **Action**

Edits the specified line

■ **Remarks**

When **EDIT** is used, BASIC types the specified program line and leaves the user in direct mode. The cursor is placed on the first character of the program line.

See your interpreter user's guide for full details on screen editing.

■ **Compiler Differences**

The **EDIT** command is not supported by the compiler.

■ **Syntax**

END

■ **Action**

Terminates program execution, closes all files, and returns control to command level

■ **Remarks**

END statements may be placed anywhere in the program to terminate execution. Unlike the **STOP** statement, **END** does not cause a "Break in line *nnnnn*" message to be printed. An **END** statement at the end of a program is optional. Microsoft GW-BASIC always returns to command level after an **END** is executed.

■ **Compiler/Interpreter Differences**

In the compiler, the **END** statement can have the following extended syntax:

END [{ DEF | SUB}]

END DEF ends a multiline function definition; you must use it with **DEF FN**. **END SUB** ends a BASIC subroutine; you must use it with **SUB**. **END DEF** and **END SUB** are supported only in the compiler.

The **END** statement in the compiler has the same effect as the sequence

```
END
SYSTEM
```

in the interpreter: it terminates program execution, closes all files, and returns control to the operating system.

The compiler always assumes an **END** statement at the end of any program, so "running off the end" (omitting an **END** statement at the end of a program) still produces proper program termination.

END Statement

■ Example

The following ends program execution if the value of the variable K is greater than 1,000; otherwise, program execution continues at line "compute".

Note the use of the line continuation character "_" to simulate a multiline **IF...THEN...ELSE**, and improve program readability.

```
if K>1000 then end _
    else goto compute
```

■ **Syntax**

ENVIRON$ (*environstring*)
ENVIRON$ (*n*)

■ **Action**

Retrieves an environment string from BASIC's environment String table

■ **Remarks**

The argument *n* is an integer.

The string result returned by the **ENVIRON$** function may not exceed 255 characters. If you specify a *environstring* name, but it either cannot be found or does not have any text following it, then **ENVIRON$** returns a null string. If you specify a *environstring* name, **ENVIRON$** returns all the associated text that follows *environstring=* in the environment string table.

If the argument is numeric, then the *n*th string in the environment string table is returned. The string in such a case, includes all the text, including the *environstring* name. If the *n*th string does not exist, a null string is returned.

ENVIRON Statement

■ **Syntax**

ENVIRON *stringexpression*

■ **Action**

Modify a parameter in MS-DOS's environment string table.

■ **Remarks**

The *stringexpression* must be of the form *parameterid=text,* or *parameterid text.* Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right, text.

If the *parameterid* has not previously existed in the environment string table, it is appended to the end of the table. If a *parameterid* exists on the table when the ENVIRON statement is executed, it is deleted and the new *parameterid* is appended to the end of the table.

The text string is the new parameter text. If the text is a null string (""), or a semicolon (";"), then the existing parameter-id is removed from the environment string table, and the remaining body of the file is compressed.

You can use this statement to change the PATH parameter for a child process, or to pass parameters to a child by inventing a new environment parameter.

Errors include parameters that are not strings and an "Out of memory" when no more space can be allocated to the environment string table. The amount of free space in the table will usually be quite small.

■ **Example**

The following operating system command will create a default PATH to the root directory on disk A:

PATH=A:\

The PATH may be changed to a new value by:

ENVIRON "PATH=A:\SALES;A:\ACCOUNTING"

A new parameter may be added to the environment string table:

```
ENVIRON "SESAME=PLAN"
```

The environment string table now contains:

```
PATH=A:\SALES;A:\ACCOUNTING
SESAME=PLAN
```

If you then entered:

```
ENVIRON "SESAME=;"
```

you would have deleted SESAME, and you would have a table containing:

```
PATH=A:\SALES;A:\ACCOUNTING
```

■ **See Also**

ENVIRON$, SHELL

EOF Function

EOF(*filenumber*)

■ **Action**

Tests for the end-of-file condition

■ **Remarks**

Returns -1 (true) if the end of a sequential file has been reached. Use **EOF** to test for end-of-file while inputting data, to avoid "Input past end" errors.

When **EOF** is used with random access files, it returns "true" if the last executed **GET** statement was unable to read an entire record because of an attempt to read beyond the end.

When you use **EOF** with a communications device, the definition of the end-of-file condition is dependent on the mode (ASCII or binary) in which you opened the device. In binary mode, **EOF** is true when the input queue is empty ($LOC(n)=0$). It becomes false when the input queue is not empty. In ASCII mode, **EOF** is false until you press CONTROL-Z, and from then on it will remain true until you close the device.

■ **Example**

```
10 OPEN "DATA" FOR INPUT AS 1
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
  .
  .
  .
```

■ **Syntax**

ERASE *arrayname [[arrayname...]]*

■ **Action**

Eliminates arrays from memory

■ **Remarks**

You can redimension arrays after erasing them, or you can use the previously allocated array space in memory for other purposes.

If the erased array is the most recently dimensioned array, **ERASE** frees space immediately. Otherwise, space is not freed until the program encounters a **FRE** command, or until a **REDIM** statement does not find sufficient space for the array or arrays it is redimensioning.

If you try to redimension an array without first erasing it, a "Duplicate definition" error will occur.

■ **See Also**

CHAIN, DIM, REDIM

■ **Compiler/Interpreter Differences**

In the compiler, **ERASE** has the same effect as in the interpreter; namely, it resets the elements of a *static* array to either zeroes or null strings, depending on the type specified by the *arrayname*.

However, executing **ERASE** on a *dynamic* array causes the array elements to be deallocated. Before your program can refer to the dynamic array again, it must first redimension the array with either a **DIM** or **REDIM** statement.

ERASE Statement

■ Example

```
rem $dynamic
dim a(100,100)
  .
  .
  .
erase a        ' This deallocates array a
redim a(5,5)
rem $static
dim b(50,50)
  .
  .
  .
erase b        ' This sets all elements of b equal to zero;
               ' b still has the dimensions assigned to it above.
```

■ **Syntax**

ERDEV
ERDEV$

■ **Action**

Provides a way to obtain device-specific status information

■ **Remarks**

ERDEV is an integer function that contains the error code returned by the last device to declare an error. **ERDEV$** is a string function which contains the name of the device driver which generated the error.

You may not set these functions.

ERDEV is set by the interrupt 24 handler when an error within the operating system is detected.

ERDEV returns the DOS INT 24 error code in its lower eight bits. The upper eight bits returned contain The following bits of the device attribute word, in the following order, are returned in the upper eight bits of the word returned by **ERDEV**: 15, 14, 13, XX, 3, 2, 1, 0. The XX value is always zero.

■ **Example**

If a user-installed device driver, MYLPT2, runs out of paper, and the driver's error number for that problem is 9, then

```
PRINT ERDEV   ERDEV$
```

prints this output:

```
9              MYLPT2
```

ERR and ERL Functions

■ **Syntax**

ERR
ERL

■ **Action**

Returns error status

■ **Remarks**

If you are using an error handling routine, the function **ERR** contains the error code for the error and the function **ERL** contains the line number of the line in which the error was found. The **ERR** and **ERL** functions are usually used in **IF...THEN** statements to direct program flow in the error handling routine.

The **RENUM** command renumbers lines when their numbers are on the *right* side of a relational operator like "=." Therefore, if you do test **ERL** in an **IF...THEN** statement, be sure to put the line number on the right side, like this:

IF ERL = 220 THEN...

With the GW-BASIC Interpreter, if the statement that causes the error is a direct mode statement, then **ERL** contains 65535. In this case, you do not want this number to be changed during a **RENUM,** so you should put it on the *left* side of a relational operator, like this:

IF 65535 = ERL THEN...

You can set **ERL** and **ERR** using the **ERROR** statement.

Because **ERL** and **ERR** are reserved words, neither may appear to the left of the equal sign in an assignment statement.

■ **Example**

The following is a test for an error in direct statement:

IF 65535 = ERL THEN PRINT "Direct Error"

When testing within a program, use:

144

```
IF ERR=error code THEN ...
    IF ERL=line number THEN ...
```

ERROR Statement

■ · Syntax

ERROR *integer-expression*

■ Action

Simulates the occurrence of a BASIC error, or allows the user to define error codes

■ Remarks

You can use **ERROR** as a statement (part of a program source line) or as a command (in direct mode).

The value of *integer-expression* must be greater than 0 and less than 256. If the value of *integer-expression* equals an error code already in use by BASIC. then the **ERROR** statement simulates the occurrence of that error and prints the corresponding error message. (See Example 1.)

To define your own error code, use a value that is greater than any used by standard GW-BASIC error codes. (Use the highest available values to maintain future compatibility with Microsoft GW-BASIC error codes.) See Example 2 for an example of a user-defined code in an error handling routine.

If an **ERROR** statement specifies a code for which no error message has been defined, the message "Unprintable error" is printed. Execution of an **ERROR** statement for which there is no error handling routine causes an error message to be printed and execution to halt.

■ Example 1

```
20 S=15
30 ERROR S
40 END
```

Since 15 is GW-BASIC error for "string too long," this prints the following:

```
String too long in line 30
```

Or, in direct mode

```
ERROR 15
```

produces:

```
String too long
```

The next example is part of a game program that allows you to make bets. The program traps the error if you exceed the "house limit." Note that the error code is 210, which BASIC doesn't use.

```
·
·
·
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B>5000 THEN ERROR 210
·
·
·
400 IF ERR=210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL=130 THEN RESUME 120
·
·
·
```

EXP Function

■ **Syntax**

EXP(x)

■ **Action**

Calculate the exponential function.

■ **Remarks**

This function returns e, which is the base of natural logarithms, to the power of x.

The exponent x must be $<= 88.02969$. If x is greater than 88.02969, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

The **EXP** function returns a single precision value unless you invoke BASIC with the **/D** switch and use a double precision variable for the argument.

■ **Example**

```
10 X=5
20 PRINT EXP(X-1)
```

This prints the following output:

```
 54.59815
```

■ **Syntax**

EXTERR(n)

■ **Action**

Returns extended error information

■ **Remarks**

EXTERR returns "extended" error information provided by versions of DOS 3.0 and greater. For versions of DOS earlier than 3.0, **EXTERR** always returns zero. The single integer argument must be in the range 0-3 as described in the table below.

Table 6.4

EXTERR function return values

n	Return Value
0	Extended error code
1	Extended error class
2	Extended error suggested action
3	Extended error locus

The values returned are *not* defined by BASIC but by DOS. Refer to the DOS Programmer's Reference (version 3.0 or later) for a description of the values returned by the DOS extended error function.

The extended error code is actually retrieved and saved by BASIC each time appropriate DOS functions are performed. Thus when an **EXTERR** function call is made, these saved values are returned.

FIELD

■ Statement Syntax

FIELD [[#]]*filenumber, fieldwidth* **AS** *string-variable...*

■ Action

Allocates space for variables in a random file buffer

■ Remarks

Before a **GET** statement or **PUT** statement can be executed, a **FIELD** statement must be executed to format the random file buffer.

The *filenumber* parameter is the number under which the file is opened. The *fieldwidth* parameter is the number of characters allocated to *string-variable*.

The total number of bytes that you allocate in a **FIELD** statement must not exceed the record length that you specified when opening the file. Otherwise, a "Field overflow" error message is generated. (The default record length is 128 bytes.)

Any number of **FIELD** statements may be executed for the same file. All **FIELD** statements that have been executed remain in effect at the same time.

All field definitions for a file are removed when the file is closed.

Note

Do not use a fielded variable name in an **INPUT** or assignment statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent **INPUT** or assignment statement with that variable name is executed, the variable's pointer no longer refers to the random record buffer, but to string space.

■ **See Also**

GET, LSET, OPEN, RSET

■ **Example 1**

This example allocates the first 20 positions (bytes) in the random file
buffer to the string variable N$, the next 10 positions to ID$, and the next
40 positions to ADD$. **FIELD** does not place any data in the random file
buffer.

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

▣ **Example 2**

Example 2 illustrates a multiply-defined **FIELD** statement. In statement
20, the 57-byte field is broken up into 5 separate variables for name,
address, city, state and zip code. In statement 30, the same field is assigned
entirely to one variable, PLIST$. Statements 60 through 90 check to see if
ZIP$, which contains the zip code, falls within a certain range; if it does,
the complete address string is printed by lines 75 and 80.

```
10    OPEN "MAILLIST" FOR RANDOM AS 1 LEN=57
20    FIELD #1, 15 AS NAM$, 25 AS ADDR$, 10 AS CTY$, 2 AS ST$, 5 AS ZI
30    FIELD #1, 57 AS PLIST$
40    GET #1, 1
50    WHILE NOT EOF (1)
60       ZCHECK$ = ZIP$
70       IF (ZCHECK$ < "85700" OR ZCHECK$ >"85800") THEN GOTO 90
75          INFO$ = PLIST$
80          PRINT INFO$
90       GET #1
100   WEND
```

И **Example 3**

This example shows the construction of a **FIELD** statement using an array
of elements of equal size:

```
, OPEN "MAILLIST" FOR RANDOM AS 1 LEN=57
.0 FOR LOOP%=0 TO 7
)0    FIELD #1,(LOOP%*16) AS OFFSET$,16 AS A$(LOOP%)
!0 NEXT LOOP%
```

FIELD

The result is equivalent to the single declaration:

```
FIELD 1,16 AS A$(0),16 AS A$(1),...,16 AS A$(6),16 AS A$(7)
```

■ Example 4

This example creates a field in the same manner as Example 3. However, the element size varies with each element:

```
5    OPEN "MAILLIST" FOR RANDOM AS 1 LEN=57
10   DIM SIZE% (4): REM Array of field sizes
20   FOR LOOP%=0 TO 4:READ SIZE%(LOOP%): NEXT LOOP%
30   DATA 9,10,12,21,41
120  DIM A$(4): REM Array of fielded variables
130  OFFSET%=0
140  FOR LOOP%=0 TO 4
150    FIELD #1,OFFSET% AS OFFSET$,SIZE%(LOOP%) AS A$(LOOP%)
160    OFFSET%=OFFSET%+SIZE%(LOOP%)
170  NEXT LOOP%
```

The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),...
SIZE%(4%) AS A$(4%)
```

■ Compiler/Interpreter Differences

The compiler does not permit fielded strings to be passed in **COMMON**.

■ **Syntax**

FILES [[*filespec*]]

■ **Action**

Prints the names of files residing on the specified disk

■ **Remarks**

The *filespec* includes either a file name or a path name and an optional device designation.

If you omit *filespec*, all the files on the currently selected drive will be listed. The *filespec* is a string formula which may contain question marks (?) or asterisks (∗) used as wild cards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks. You don't need to use the asterisk when you are asking for all the files on a drive. For example:

```
FILES "B:"
```

If you use a *filespec* without an explicit path, the current directory is the default.

■ **Examples**

Shows all files on the current directory:

```
FILES
```

Shows all files with the extension .BAS:

```
FILES "*.BAS"
```

Shows all files on drive B:

```
FILES "B:*.*"
```

Equivalent to "B:∗.∗":

```
FILES "B:"
```

FILES Statement

Shows all five-letter files whose names start with "TEST" and end with the .BAS extension:

```
FILES "TEST?.BAS"
```

If SALES is a subdirectory of the current directory, this statement displays SALES *dir*. If SALES is a file in the current directory, the following statement displays SALES.

```
FILES "\SALES"
```

Displays MARY *dir* if MARY is a subdirectory of SALES (if MARY is a file, displays its name):

```
FILES "\SALES\MARY"
```

■ **Syntax**

FIX(x)

■ **Action**

Returns the truncated integer part of x

■ **Remarks**

FIX(x) is equivalent to SGN(x)∗INT(ABS(x)). The difference between FIX and INT is that, for negative x, FIX returns the first negative integer greater than x, while INT returns the first negative integer less than x.

See the "INT Function" and the "CINT Function" for other functions that return integer values.

■ **Example 1**

```
PRINT FIX(58.75)
```

This prints

```
 58
```

■ **Example 2**

```
PRINT FIX(-58.75)
```

This prints

```
-58
```

FOR...NEXT Statements

◙ Statement Syntax

FOR *counter* = *start* **TO** *end* [[**STEP** *increment*]]

.
.
.

NEXT [[*counter*]][[*,counter...*]]

■ Action

Allows a series of instructions to be performed in a loop a given number of times

■ Remarks

The **FOR** statement uses *start, end,* and *increment* as fixed numeric expressions, and *counter* as a counter. The expression *start* is the initial value of the counter. The expression *end* is the final value of the counter. The program lines following the **FOR** statement are executed until the **NEXT** statement is encountered. Then *counter* is adjusted by the amount specified by **STEP**, and compared with the final value, *end*. (If you do not specify **STEP**, the increment is assumed to be one.) If *counter* is still not greater than *end*, then BASIC branches back to the statement after the **FOR** statement and the process is repeated. If it is greater, execution continues with the statement following the **NEXT** statement.

If **STEP** is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

It is a good idea not to change the loop variable within the loop, since doing so can make the program more difficult to debug.

■ Nested Loops

You may nest **FOR...NEXT** loops; that is, you may place a **FOR...NEXT** loop within another **FOR...NEXT** loop. When loops are nested, each loop must have a unique variable name as its counter. The **NEXT** statement for the inside loop must appear before the **NEXT** statement for the outside loop. This construction is the correct form:

```
FOR I = 1 TO 10
   FOR J = 1 TO 10
```

```
      FOR K = 1 TO 10
         .
         .
         .
      NEXT K
    NEXT J
  NEXT I
```

A **NEXT** statement that has the form,

```
NEXT K, J, I
```

is equivalent to the sequence of statements

```
NEXT K
NEXT J
NEXT I
```

If you omit the variable in a **NEXT** statement, the **NEXT** statement matches the most recent **FOR** statement. If a **NEXT** statement is encountered before its corresponding **FOR** statement, a "NEXT without FOR" error message is generated and execution is terminated.

■ Example 1

```
10 for i= 5 to 1 step -1
20    for j= 1 to i
30      print "*";
40    next j
50    print
60 next I
```

Output:

```
* * * * *
* * * *
* * *
* *
*
```

FOR...NEXT Statements

■ Compiler/Interpreter Differences

The GW-BASIC Compiler supports double precision control values *end*, and *counter*) in its **FOR...NEXT** loops.

■ Example 2

This example prints a sine curve. It should be compiled with the /N switch if it is run as is (without line numbers).

```
cls
locate 12,1
print string$(80,95)
for i! = 0 to 6.3 step .07875
    column% = 12.65*i! + 1
    row% = 11*(1 - sin(i!)) +1
    locate row%,column%
    print chr$(222)
next
```

■ **Syntax**

FRE(*number*)
FRE(*string-exp*)

■ **Action**

Returns the size of free string space

■ **Remarks**

In interpreted BASIC programs, **FRE** with a numeric argument returns the number of bytes of memory not being used by BASIC.

In both compiled and interpreted BASIC programs, **FRE** with a string argument returns the number of bytes in BASIC's memory space that are not being used; however, before it returns the number of free bytes, it firsts compacts free string space into a single block.

■ **Example**

PRINT FRE (0)

Sample output:

 59530

■ **Compiler/Interpreter Differences**

In compiled programs, **FRE** with a numeric argument returns the size of the next free block of string space.

GET Statement - Graphics

■ **Syntax**

$GET(x1,y1)-(x2,y2),array\text{-}name$

.

.

.

$PUT(x1,y1)$, *array name* [[,*action verb*]]

■ **Action**

Transfers graphic images to and from the screen

■ **Remarks**

The **GET** statement transfers a screen image into *arrayname*. This image is bounded by a rectangle that is defined by the specified points.

The **PUT** statement transfers the image stored in the array onto the screen. The range $(x1,y1)-(x2,y2)$ is a rectangular area on the display screen. The coordinates $(x1,y1)$ is the upper left-hand corner and the coordinates $(x2,y2)$ is the lower right-hand corner.

The *arrayname* is the name assigned to the place that will hold the image. The array can be any type except string. Its dimensions must be large enough to hold the entire image. Unless the array type is integer, the contents of an array after a **GET** will be meaningless when interpreted directly.

One of the most useful things that can be done with **GET** and **PUT** is animation. (See the "PUT Statement" for a discussion of animation.)

■ **Syntax**

GET [# [*file-number*[,*record-number*]]

■ **Action**

Reads a record from a random disk file into a random buffer

■ **Remarks**

The *filenumber* is the number under which you opened the file. If you omit *recordnumber*, the next record (the one after the last **GET**) is read into the buffer. The largest possible record number is 16,777,215.

The **GET** and **PUT** statements allow fixed-length input and output for GW-BASIC COM files. However, because of the low performance associated with telephone line communications, it is recommended that you do not use **GET** and **PUT** for telephone communication.

■ **Example**

```
GET #1,75
```

This reads record number 75 from file number 1.

Note

You may use **INPUT#** and **LINE INPUT#** after a **GET** statement to read characters from the random file buffer.

You may use the **EOF** function after a **GET** statement to see if that **GET** was beyond the end of file marker.

GOSUB...RETURN Statements

▨ Syntax

GOSUB *linenumber1*

.

.

.

RETURN [*linenumber2*]

■ Action

Branches to, and returns from, a subroutine

■ Remarks

The *linenumber1* in the GOSUB statement is the first line of the subroutine.

In addition to the simple RETURN statement, the RETURN *linenumber2* option allows a RETURN from a GOSUB statement to the statement having the specified line number, instead of a normal return to the statement following the GOSUB statement. Use this type of return with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the GOSUB will remain active, and errors such as "FOR without NEXT" may result. Additionally, event-trapping routines should never use the RETURN *linenumber2* option unless only one event can occur, because an event occurring during an event-trapping routine will leave confusing information on the stack, and this will result in undefined program behavior.

You can call a subroutine any number of times in a program. You can also call a subroutine from within another subroutine. Such nesting of subroutines is limited only by the amount of stack space, which can be changed using the CLEAR statement. If the stack space is increased by using the CLEAR statement, then the maximum number of nested GOSUB statements is also increased.

A subroutine may contain more than one RETURN statement. A simple RETURN statement (without the *linenumber2* option) in a subroutine causes BASIC to branch back to the statement following the most recent GOSUB statement.

Subroutines may appear anywhere in the program, but it is good programming practice to make them readily distinguishable from the main program. To prevent inadvertent entry into a subroutine, precede it with a **STOP, END,** or **GOTO** statement that directs program control around the subroutine.

■   **Compiler/Interpreter Differences**

The compiler supports line labels for statements, so the enhanced compiler syntax is as follows:

GOSUB { *linenumber1*¦*linelabel1*}
.
.
.
RETURN [[{ *linenumber2*¦*linelabel2*}]]

Note

> The preceding discussion of subroutines applies only to the targets of **GOSUB** statements, *not* subroutines delimited by **SUB...END SUB.**

GOSUB...RETURN Statements

■ **Example**

This example computes arcsine values for arguments betweeen −1 and 1. This is the inverse of the sine function, so the value returned will be an angle whose measure is between $\pi/2$ radians and $3\pi/2$ radians (90° to 270°).

Compile this example with the /N switch to turn off the line-numbering constraint.

```
input "sine";x
while (x < -1 or x > 1)
   print "Illegal sine value."
   print "Sine must be >= -1 and =< 1."
   input "sine";x
wend
pi = 3.141593
guess2 = pi/2 : guess1 = 3 * (pi/2)
while abs(guess1 - guess2) > .0000005
   gosub newval
wend
print "arcsin ";x;" = "; temp
end


newval:
   temp = (guess1 + guess2)/2
   if sin(temp) > x then guess2 = temp _
      else guess1 = temp
return
```

Sample output:

```
sine? 3
Illegal sine value.
Sine must be >= -1 and =< 1.
sine? .43
arcsin   .43 = 2.6971
```

That is, the angle whose sine is .43 has a measure of 2.6971 radians.

■ **Statement Syntax**

GOTO *linenumber*

■ **Action**

Branches unconditionally to the line specified by *linenumber*

■ **Remarks**

If the statement that has *linenumber* is an executable statement, execution continues with that statement. If it is a nonexecutable statement, such as a **REM** or **DATA** statement, execution proceeds at the first executable statement encountered after *linenumber*.

It is good programming practice to use structured control statements (**IF...THEN...ELSE, WHILE...WEND**) instead of **GOTO** statements as a way of branching, because a program with many **GOTO** statements can be difficult to read and debug.

■ **Compiler/Interpreter Differences**

In the compiler, the **GOTO** statement allows branching to lines identified by either line numbers or line labels.

The enhanced compiler syntax is as follows:

GOTO { *linenumber* ¦ *linelabel*}

GOTO

■ Example

The following program prints the area of the circle that has the input radius:

```
print "Input 0 to end."
start:
    input r
    if r <= 0 then end _
    else _
        a = 3.14 * r^2
        print "Area =";a
goto start
```

Sample output:

```
Input 0 to end.
? 5
Area = 78.5
? 7
Area = 153.86
? 12
Area = 452.16
? 0
```

■ **Syntax**

HEX\$ (*expression*)

■ **Action**

Returns a string that represents the hexadecimal value of the decimal argument *expression*

■ **Remarks**

The argument *expression* is rounded to an integer before HEX\$ evaluates it.

■ **Example**

```
10 INPUT X
20 A$=HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
```

Output:

```
? 32
  32 DECIMAL IS 20 HEXADECIMAL
```

IF...THEN / IF...GOTO Statements

- **Syntax**

IF *expression*[,] **THEN** *target* [[,] **ELSE** *alternative*]
IF *expression*[,] [**THEN**] **GOTO** *target* [[,] **ELSE** *alternative*]

- **Action**

Makes a decision regarding program flow based on the result returned by an expression

- **Remarks**

The entire **IF...THEN...ELSE** must be on *one* logical line.

If the result of *expression* is true (or if it is not equal to zero), the *target* is executed. **THEN** may be followed by either a line number for branching or one or more statements to be executed. **GOTO** is always followed by a line number.

If the result of *expression* is false (or if it is equal to zero), the *target* clause is ignored and the **ELSE** *alternative,* if present, is executed.

If you omit the **ELSE** *alternative,* program execution continues with the next executable statement. You can put a comma before **THEN** for readability.

- **Nesting of IF Statements**

You may include **IF...THEN** statements inside other **IF...THEN** statements. This nesting is limited only by the length of the line. For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF X<Y _
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of **ELSE** and **THEN** clauses, each **ELSE** is matched with the closest unmatched **THEN.** For example:

```
IF A=B THEN IF B=C THEN PRINT "A=C" _
     ELSE PRINT "A<>C"
```

prints

A<>C

when the following conditions are true:

A=B
B<>C

However, the statement above does *not* print "A<>C" when

A<>B
B=C

Instead, the program simply continues with the next executable statement. If you wanted your program to print A<>C when A<>B and B=C, then your statement would look like this:

```
IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT "A<>C" _
       ELSE IF B=C THEN PRINT "A<>C"
```

If you enter a line number after an **IF...THEN** statement while in direct mode, an "Undefined line" error results, unless you have previously entered a statement with the specified line number in indirect mode.

Note

When using **IF** to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test is true if the difference between A and 1.0 is less than .000001.

IF...THEN / IF...GOTO Statements

■ Example 1

This statement gets record number I if I is not zero:

```
200 IF I THEN GET#1,I
```

■ Example 2

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, execution branches to line 130. If I is not in this range, execution continues with line 110.

```
100 IF (I<20 AND I>10) THEN 130
110 PRINT "OUT OF RANGE"
120 END
130 DB = DB * I
140 I = I + 1
 .
 .
 .
```

■ Example 3

This statement in a program prints output on the screen or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

■ Compiler/Interpreter Differences

The compiler allows indefinite line continuation with the underscore character, allowing you to simulate multiline **IF...THEN...ELSE** statements. As a result, fully nested **IF...THEN...ELSE** control structures can be set up by using extra-long statements.

Also, since the compiler supports line labels, you can use this syntax:

IF *expression* [[THEN]] **GOTO** *linelabel*

170

■ **Syntax**

INKEY$

■ **Action**

Returns either a one-character string containing a character read from the standard input device or a null string if no character is pending there.

■ **Remarks**

The keyboard is usually the standard input device. No characters are echoed. All characters are passed through to the program except for Break, which terminates the program.

■ **Example**

The following example is a subroutine that, checks to see, for the interval TIMELIMIT%, if there has been any input from the keyboard. If there hasn't been any input, the subroutine continues until TIMELIMIT% is exceeded. If there has, line 1040 checks to see if the input is ASCII character 13, the carriage return. If it is, TIMEOUT% is set to zero and control is returned to the main program.

```
1000 REM **TIMED INPUT SUBROUTINE**
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030    A$=INKEY$
1035    IF LEN(A$)=O THEN 1060
1040    IF ASC(A$)=13 THEN TIMEOUT%=O
1045    IF TIMEOUT% = O THEN RETURN
1050    RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

■ **Compiler/Interpreter Differences**

The compiler may read a **BREAK** if the /D option is not specified at compile time.

INP Function

■ **Syntax**

INP(*port*)

■ **Action**

Returns the byte read from *port*. The *port* must be an integer in the range 0 to 65535.

■ **Remarks**

INP is the complementary function to the **OUT** statement.

■ **See Also**

OUT

■ **Example**

This instruction reads a byte from port 54321 and assigns it to the variable A:

```
100 A=INP(54321)
```

In 8086 assembly language, this is equivalent to:

```
MOV DX,54321
IN  AL,DX
```

■ **Syntax**

INPUT[;][*"prompt"*;] *variable1*[, *variable2*]...

■ **Action**

Allows input from the keyboard during program execution

■ **Remarks**

When the program encounters an **INPUT** statement, it pauses and prints a question mark to indicate that it is waiting for data. If you include *prompt*, it prints this before the question mark. You can then enter the required data at the keyboard.

BASIC can be re-directed to read from standard input and write to standard output by providing the input and output file names when invoking BASIC.

You may use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, the statement

```
INPUT "ENTER BIRTHDATE", B$
```

prints the prompt with no question mark.

If **INPUT** is immediately followed by a semicolon, then the RETURN that you type to input data does not echo a carriage return/linefeed sequence.

The data that you enter is assigned to the variable(s) in the statement. The number of data items that you supply must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that you input must agree with the type specified by the variable name. (Strings input to an **INPUT** statement need not be surrounded by quotation marks.)

Responding to **INPUT** with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until you give an acceptable response.

INPUT Statement

■ Example 1

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
```

Output:

```
? 5

  5 SQUARED IS 25
```

■ Example 2

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 IF R=-1 THEN END
40 A=PI*R^2
50 PRINT "THE AREA OF THE CIRCLE IS";A : PRINT
60 GOTO 20
```

Output:

```
WHAT IS THE RADIUS?   7.4

THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?   -1
```

174

■ Syntax

INPUT# *filenumber, variable1[, variable2]...*

■ Action

Reads data items from a sequential device or file and assigns them to program variables

■ Remarks

The *filenumber* is the number used when the file is opened for input. The variable list contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) Unlike **INPUT, INPUT#** does not print a question mark.

The data items in the file should appear just as they would if you were entering data in response to an **INPUT** statement. With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If GW-BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. This means a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or linefeed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being **INPUT**, the item is terminated.

■ Example

INPUT#2,A,B,C

INPUT$ Function

■ Syntax

INPUT$(x[,[#]filenumber])

■ Action

Returns a string of x characters read from *filenumber*.

■ Remarks

If the *filenumber* is not specified, the characters will be read from the standard input device. (If input has not been redirected, the keyboard is the standard input device).

If the keyboard is used for input, no characters will be echoed on the screen. All control characters are passed through except Break, which is used to interrupt the execution of the INPUT$ function.

BASIC can be redirected to read from standard input by providing the input filename on the command line.

■ Example 1

This lists the contents of a sequential file in hexadecimal:

```
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

■ Example 2

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
 .
 .
 .
```

176

■ **Syntax**

INSTR($[\![start,]\!]x\$,y\$$)

■ **Action**

Searches for the first occurrence of string $y\$$ in $x\$$, and returns the position at which the match is found

■ **Remarks**

The optional offset *start* sets the position for starting the search; *start* must be in the range 1 to 255.

The arguments $x\$$ and $y\$$ may be string variables, string expressions, or string literals.

Condition	What INSTR Returns
start greater than length of $x\$$	0
$x\$$ is null string	0
$y\$$ cannot be found	0
$y\$$ is null string	*start* or 1
no *start* specified	1

Use the **LEN** function to find the length of $x\$$.

■ **Example**

```
10 X$="ABCDEB"
20 Y$="B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
```

Output:

```
 2   6
```

INT Function

■ **Syntax**

INT(x)

■ **Action**

Returns the largest integer less than or equal to x

■ **Example 1**

```
PRINT INT(99.89), INT(-12.11)
```

Output:

```
 99      -13
```

■ **See Also**

CINT, FIX

■ **Syntax**

IOCTL$ ([#]*filenumber*)

■ **Action**

Receives a control data string from a device driver

■ **Remarks**

The **IOCTL$** function is most frequently used to receive acknowledgement that an **IOCTL** statement succeeded or failed, or to obtain current status information.

You could use **IOCTL$** to ask a communications device to return the current baud rate, information on the last error, logical line width, and so on.

The **IOCTL$** function works only if:

1. The device driver is installed.
2. The device driver states it processes **IOCTL** strings.
3. BASIC performs an **OPEN** on a file on that device.

■ **See Also**

IOCTL

■ **Example**

This example tells the device that the data is raw:

```
10 OPEN "\DEV\ENGINE" AS 1
20 IOCTL #1, "RAW"
```

In this continuation, if the character driver ENGINE responds "false" from the raw data mode **IOCTL** statement, then the file is closed:

```
30 IF IOCTL$(1) = "0" THEN CLOSE 1
```

IOCTL Statement

▨ Syntax

IOCTL [[#]]*filenumber, string*

▨ Action

Transmits a control character or string to a device driver

▣ Remarks

IOCTL commands are generally two to three characters followed optionally by an alphanumeric argument. An **IOCTL$** command string may be up to 255 bytes long.

The **IOCTL** statement works only if:

1. The device driver is installed.
2. The device driver states it processes **IOCTL** strings.
3. BASIC performs an **OPEN** on a file on that device.

Most standard MS-DOS device drivers don't process **IOCTL** strings, and it is necessary for you to determine if the specific driver can handle the command.

▨ See Also

IOCTL$

▣ Example

If you wanted to set the page length to 66 lines per page on LPT1, your procedure might look like this:

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS 1
20 IOCTL$ 1, "PL66"
```

■ **Syntax**

KEY *n*, *x*$
KEY LIST
KEY ON
KEY OFF

■ **Action**

Assigns soft key values to function keys and displays the values

■ **Remarks**

The argument *n* is the number of the function key.

The argument *x*$ is the text assigned to the specified key.

The **KEY** statement allows you to designate special "soft key" functions for the function keys by assigning each of the function keys a 15-byte string which is input to BASIC when you press that key.

You can display softkeys with the **KEY ON**, **KEY OFF**, and **KEY LIST** statements.

KEY ON causes the softkey values to be displayed on the bottom line of the screen.

KEY OFF erases the softkey display from the bottom line, making that line available for program use. It does not disable the function keys.

KEY LIST displays all softkey values on the screen, with all 15 characters of each key displayed.

Assigning a null string (string of length 0) to a softkey disables the function key as a softkey.

If the function key number is not in the range of permissible function key numbers, an "Illegal function call" error is displayed, and the previous key string expression is retained.

KEY Statement

When a softkey is assigned, the **INKEY$** function returns one character of the softkey string per invocation.

◙ Examples

Displays the softkeys on bottom line of screen:

```
50 KEY ON
```

Erases softkey display:

```
60 KEY OFF
```

Assigns the string

```
"MENU"
```

followed by a carriage return to soft key 1:

```
70 KEY 1,"MENU"+CHR$(13)
```

Disables soft key 1:

```
80 KEY 1,""
```

The following routine initializes the first five soft keys. (Line 10 turns off the key display during initialization; line 70 displays the new softkey values.)

```
10 KEY OFF
20 DATA "EDIT ","LET ","SYSTEM","PRINT ","LPRINT "
30 FOR I = 1 TO 5
40   READ SOFTK$(I)
50   KEY I,SOFTK$(I)
60 NEXT I
70 KEY ON
```

◙ Compiler/Interpreter Differences

The compiler does not preserve soft key string values across chains with the **CHAIN** statement.

■ **Syntax**

KEY(*n*) ON
KEY(*n*) OFF
KEY(*n*) STOP

■ **Action**

Activates or deactivates trapping of the specified key.

■ **Remarks**

The argument *n* is the number of a function key, a cursor direction key, or a user-defined key. (See "KEY Statement" for information on assigning softkey values to function keys.)

Value of *n*	Key
1 - 10	The function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-25	User-defined keys
30	F11
31	F12

User-defined keys are defined by the statement:

KEY *n*,CHR\$(*j*)+CHR\$(*k*)

where *n* is the user key number. The arguments *j* and *k* are defined by the hardware and uniquely define a key on the keyboard.

Note that the **KEY** statement described in the previous section assigns softkey and cursor direction values to function keys and displays these values. Do not confuse **KEY ON** and **KEY OFF**, which display and erase these values, with the event trapping statements described in this section.

KEY(n) Statement

The **KEY(n) ON** statement enables softkey or cursor direction key event trapping by an **ON KEY** statement. If you specify a non-zero line number in the **ON KEY** statement while trapping is enabled, BASIC checks between every statement to see if you have pressed **KEY(n)**. If you have, it executes the **GOSUB** in the **ON KEY** statement. The text that would normally be associated with a function key is not printed.

KEY(n) OFF disables the event trap. Even if an event takes place, it is not remembered.

KEY(n) STOP disables the event trap, but, if you press the specified key, your action is remembered and an **ON KEY** statement is executed as soon as a **KEY(n)** statement is executed.

■ **Example**

```
10 KEY 4,SCREEN 0,0 ' assigns soft key 4
20 KEY(4) ON         'enables event trapping
 .
 .
 .
70 ON KEY(4) GOSUB 200
 .
 .
 .

(key 4 pressed)

 .
 .
 .
200 'Subroutine for screen
```

■ **Compiler/Interpreter Differences**

See compiler note under "ON KEY Statement."

■ **Syntax**

KILL *filespec*

■ **Action**

Deletes a file from disk

■ **Remarks**

KILL is similar to the operating system **ERASE** command. If a **KILL** statement is given for a file that is currently open, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random dat ı files, and sequential data files. The *filespec* may contain question marks (?) or asterisks (∗) used as wildcards. A question mark matches any singlı: character in the file name or extension. An asterisk matches one or morı. characters starting at its position.

Since it is possible to refer to the same file in a sub-directory via different paths, it is nearly impossible for BASIC to know that it is indeed the same file simply by looking at the path. For example if MARY is your current directory, then all of the following refer to the same file:

```
"REPORT"
"\SALES\MARY\REPORT"
"..\MARY\REPORT"
"..\..\MARY\REPORT"
```

Therefore, any open file with the same file name will cause a "file already open" error.

You can use **KILL** only to delete files. To delete directories, use the **RMDIR** command.

Warning

Be extremely careful when using wildcards with **KILL**.

KILL Statement

■ Examples

The position taken by the question mark matches any valid filename character. This command kills any file that has a six-character-name starting with DATA1 and has the file name extension .DAT. This includes DATA1O.DAT and DATA1Z.DAT.

```
200 KILL "DATA1?.DAT"
```

Kills all files named DATA1, regardless of the file name extension:

```
210 KILL "DATA1.*"
```

Kills all files with the extension .DAT in a directory called GREG:

```
220 KILL "..\GREG\*.DAT"
```

■ **Syntax**

LEFT$$(x$,n)$

■ **Action**

Returns a string comprising the leftmost n characters of x

■ **Remarks**

The integer argument n must be in the range 0 to 255.

If n is greater than the number of characters in x, the entire string is returned. (To find the number of characters in x, use LEN$(x$)$.

If $n = 0$, the null string (length zero) is returned.

■ **Example**

This prints the leftmost 5 characters of A$:

```
10 A$="BASIC LANGUAGE"
20 B$=LEFT$(A$,5)
30 PRINT B$
```

Output:

```
BASIC
```

■ **See Also**

MID$, RIGHT$

LEN Function

■ Syntax

LEN($x\$$)

■ Action

Returns the number of characters in $x\$$

■ Remarks

Nonprinting characters and blanks are counted.

■ Example

```
10 X$="PORTLAND, OREGON"
20 PRINT LEN(X$)
```

Output:

16

■ **Syntax**

[[LET]] *variable=expression*

■ **Action**

Assigns the value of an expression to a variable

■ **Remarks**

Notice that the word **LET** is optional; that is, the equal sign is sufficient for assigning an expression to a variable name.

■ **Example**

Corresponding lines perform the same function in these two examples:

```
110 LET D=12
120 LET E=12-2
130 LET F=12-4
140 LET SUM=D+E+F
 .
 .
 .
```

or

```
110 D=12
120 E=12-2
130 F=12-4
140 SUM=D+E+F
 .
 .
 .
```

Use of **LET** is archaic.

LINE Statement

■ **Syntax**

LINE [[[STEP] (*x1,y1*)]-[STEP] (*x2,y2*) [,[*color*][,b[f]]][,*style*]

■ **Action**

Draws a line or box on the screen.

■ **Remarks**

The coordinate is the starting point of the line.

The coordinate is the ending point for the line.

The **STEP** option makes the specified coordinates relative to the "most recent point," instead of absolute, mapped coordinates. For example, if the most recent point referred to by the program is (10,10), then

```
LINE STEP (10,5)
```

refers to a point with x-coordinate 10 + 10, and y-coordinate 10 + 5, or (20,15).

If the **STEP** option is used for the second coordinate on a **LINE** statement, it is relative to the first coordinate in the statement. Other ways to establish a new *most recent point* are to initialize the screen with the **CLS** and **SCREEN** statements. Using the **PSET, PRESET, CIRCLE** and **DRAW** statements will also establish a new "most recent point."

The *color* is the number of the color in which the line is drawn. (If the ,b or ,bf option is used, the box is drawn in this color.)

The ,b option draws a box with the points and specifying the upper left and lower right corners.

The ,bf option draws a filled box. This option means draw a box like the one defined by ,b, but also fill in the interior points with the selected color.

The *style* is a 16-bit integer mask used to put pixels on the screen. This is called "line styling." Each time **LINE** plots a point on the screen, it uses the current circulating bit in *style*. If that bit is a 0, then no point is plotted; if the bit is a 1, a point is plotted. After plotting a point, **LINE** selects the next bit position in *style*.

190

Since a 0 bit in *style* causes no change to the point on the screen, you may prefer to draw a background line before a "styled" line in order to force a known background. Style is used for normal lines and boxes, but has no effect on filled boxes.

When coordinates specify a point that is not in the current viewport, the line segment is clipped to the viewport.

■ Examples

The following examples assume a screen 320 pixels wide by 200 pixels high.

Draws a line in the foreground color from the most recent point to x2,y2:

```
10 LINE-(x2,y2)
```

Draws a diagonal line across the screen (downward):

```
20 LINE (0,0) - (319,199)
```

Draws a line across the screen:

```
30 LINE (0,100) - (319,100)
```

Draws a line in color 2:

```
40 LINE (10,10) - (20,20),2
```

Draws an alternating line on/line off pattern on a monochrome display:

```
10 FOR x=0 to 319
20 LINE (x,0) - (x,199),x AND 1
30 NEXT
```

Draws a box in the foreground (note that the color is not included):

```
10 LINE '(0,0) - (100,100),,b
```

Draws a filled box in color 2 (coordinates are given as offsets with the **STEP** option):

```
20 LINE STEP (0,0) -STEP (200,200),2,bf
```

Draws a dashed line from the upper left hand corner to the center of the screen:

LINE Statement

```
10 LINE (0,0) - (160,100) , 3, , &HFF0C
```

■ **Syntax**

LINE INPUT[;] [**"promptstring"**;] *stringvariable*

■ **Action**

Inputs an entire line (up to 254 characters) to a string variable, without the use of delimiters

■ **Remarks**

The *promptstring* is a string literal that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of *promptstring*. All input from the end of *promptstring* to the carriage return is assigned to *stringvariable*. However, if a linefeed/carriage return sequence (this order only) is encountered, both characters are echoed; however, the carriage return is ignored, the linefeed is put into *stringvariable*, and data input continues.

If **LINE INPUT** is immediately followed by a semicolon, then the carriage return that you enter to end the input line does not echo a carriage return/linefeed sequence on the screen.

You may abort a **LINE INPUT** statement by pressing CONTROL-C. GW-BASIC then returns to command level. If you are using the interpreter, entering **CONT** resumes execution at the **LINE INPUT**.

■ **Example**

See the example under the "LINE INPUT# Statement."

LINE INPUT# Statement

■ **Syntax**

LINE INPUT# *filenumber, stringvariable*

■ **Action**

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable

■ **Remarks**

The *filenumber* is the number under which the file was opened. The *stringvariable* is the variable name to which the line will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return. If a linefeed/carriage return sequence is encountered, it is preserved. In other words, the linefeed/character return characters are returned as part of the string.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format is being read as data by another program.

When GW-BASIC is invoked with redirected input and output, all LINE INPUT statements will read from the specified input file instead of the keyboard.

When input is redirected, GW-BASIC will continue to read from this source until detecting a CONTROL-Z. This condition may be tested with the EOF function. If the file is not terminated by a CONTROL-Z, or a BASIC file input statement tries to read past end-of-file, then any open files are closed, the message "Read past end" is written to standard output, and BASIC returns to the operating system.

■ **Example**

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
```

```
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
```

Output:

CUSTOMER INFORMATION? **LINDA JONES   234,4   MEMPHIS**

LINDA JONES   234,4   MEMPHIS

LIST Command

■ **Syntax**

LIST [[*linenumber*]] [[-[[*linenumber*]]]] [[,*device*]]

■ **Action**

Lists all or part of the program currently in memory

■ **Remarks**

The *linenumber* is in the range 0 to 65529.

The *device* is a device designation string, such as SCRN: or LPT:, or a file name.

■ **Description**

If *linenumber* is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing BREAK.) If *linenumber* is included, only the specified line will be listed. BASIC always returns to command level after a LIST is executed.

If only the first *linenumber* is specified, that line and all higher-numbered lines are listed.

If only the second *linenumber* is specified, all lines from the beginning of the program through that line are listed.

If both *linenumbers* are specified, the entire range is listed.

If the *device* is omitted, the listing is displayed at the terminal.

■ **Examples**

Lists the program currently in memory:

LIST

Lists line 500:

LIST 500

Lists all lines from 150 to the end:

`LIST 150-`

Lists all lines from the lowest number through 1000:

`LIST -1000`

Lists lines 150 through 1000, inclusive:

`LIST 150-1000`

Lists lines 150 through 1000 on the line printer:

`LIST 150-1000,"LPT:"`

■ **Compiler Differences**

The **LIST** command is not supported by the compiler:

LLIST Command

■ Syntax

LLIST [*linenumber*[-[*linenumber*]]]

■ Action

Lists all or part of the program currently in memory on the line printer

■ Remarks

LLIST assumes a 132-character-wide printer.

BASIC always returns to command level after an **LLIST** is executed. The options for **LLIST** are the same as for **LIST**.

■ Example

See the examples for under **LIST**.

■ Compiler Differences

The **LLIST** command is not supported by the compiler.

■ **Syntax**

LOAD *filespec*[[,R]]

■ **Action**

Loads a file from an input device into memory

■ **Remarks**

For loading a program, the *filespec* is an optional device specification followed by a filename or pathname that conforms to operating system naming conventions. BASIC appends the default filename extension .**BAS** if the user specifies no extensions, when the file is saved to the disk.

The *filespec* must include the file name that was used when the file was saved, or created by an editor. (BASIC will append a default file name extension if one was not supplied in the **SAVE** command.)

The **R** option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the **R** option is used with **LOAD**, the program is run after it is loaded, and all open data files are kept open. Thus, **LOAD** with the **R** option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

■ **Example**

Loads and runs the program STRTRK.BAS:

LOAD "STRTRK",R

Loads the program MYPROG.BAS from the disk in drive B, but does not run the program:

LOAD "B:MYPROG"

LOAD Command

■ **Compiler Differences**

The **LOAD** command is not supported by the compiler.

■ **Syntax**

LOC(*filenumber*)

■ **Action**

Returns the current position within the file

With random disk files, **LOC** returns the actual record number within the file.

With sequential files, **LOC** returns the current byte position in the file, divided by 128.

■ **Remarks**

The *filenumber* is the number under which the file was opened.

When a file is opened for **APPEND** or **OUTPUT**, **LOC** returns the size of the file in 128-byte blocks.

For a communications file, **LOC**(*filenumber*) is used to determine if there are any characters in the input queue waiting to be read. If there are more than 255 characters in the queue, **LOC**(*filenumber*) returns 255. Since interpreter strings are limited to 255 characters, you do not need to test for string size before reading data into it.

If fewer than 255 characters remain in the queue, the value returned by **LOC**(*X*) depends on whether the device was opened in ASCII or binary mode. In either mode, **LOC** will return the number of characters that can be read from the device. However, in ASCII mode, the low level routines stop queueing characters as soon as end-of-file is received. The end-of-file itself is not queued and cannot be read. An attempt to read the end-of-file will result in an "Input past end" error.

■ **Example**

```
200 IF LOC(1)>50 THEN STOP
```

LOCATE Statement

■ Syntax

LOCATE[[row]][,[col]][,[cursor]][,[start]][,stop]]]]]

■ Action

Moves the cursor to the specified position

■ Remarks

The *row* is a line number (vertical) on the screen. The *row* is a numeric expression returning an unsigned integer.

The *col* is the column number on the screen. It is a numeric expression returning an unsigned integer.

Optional parameters turn the blinking cursor on and off and define the vertical start and stop lines.

The *cursor* is a Boolean value indicating whether the cursor is visible or not. A value of 0 (zero) indicates cursor *off*, a value of 1 indicates cursor *on*.

The *start* is the cursor starting line (vertical) on the screen. It should be a numeric expression returning an unsigned integer.

The *stop* is the cursor stop line (vertical) on the screen. It should be a numeric expression returning an unsigned integer.

Any value outside the specified ranges will result in an "Illegal function call" error. In this case, previous values are retained.

You may omit any parameter from this statement. If you do this, the program assumes the previous value for the parameter.

Note that the *start* and *stop* lines are the CRT scan lines that specify which pixels on the screen are lit. A wider range between the start and stop lines will produce a taller cursor, such as one that occupies an entire character block.

If the *start* line is given but the *stop* line is omitted, *stop* assumes the same value as *start*.

The last line on the screen is reserved for softkey display and is not accessible to the cursor unless the softkey display is off and **LOCATE** is used to get to it.

■   **Examples**

Moves cursor to upper-left corner of the screen:

10 LOCATE 1,1

Makes the cursor visible; position remains unchanged:

20 LOCATE,,1

Position and cursor visibility remain unchanged (sets the cursor to display at the bottom of the character starting and ending on raster line 7):

30 LOCATE,,,7

Moves the cursor to line 5, column 1; turns cursor on (cursor will cover entire character cell starting at scan line 0 and ending on scan line 7): •

40 LOCATE 5,1,1,0,7

LOCK...UNLOCK Statements

■ Syntax

LOCK [[#]] *filenum* [[,{ *record* ¦ [[*start*]] **TO** *end*}]]
.
.
.

UNLOCK [[#]] *filenum* [[,{ *record* ¦ [[*start*]] **TO** *end*}]]

■ Action

Controls access by other processes to all or part of an opened file

■ Description

They are used in networked environments where several users might be working simultaneously on the same file.

■ Remarks

The *filenum* is the number with which the file was opened. If you specify just one *record* then only that record is locked or unlocked. If you specify a range of records and omit a starting record then all records from the first record to the end of the range are locked or unlocked. **LOCK** with no record arguments locks the entire file, while **UNLOCK** with no record arguments unlocks the entire file. A *record* may be any integer from 1 to 4,294,967,295 (i.e., $2^32 - 1$). A record may be up to 32,767 bytes in length.

If the file has been opened for *random* input or output, then the range indicates which records are to be locked or unlocked. However, if the file has been opened for sequential input or output, **LOCK** and **UNLOCK** affect the entire file, regardless of the range specified by *start* to *end*.

Note

> Be sure to remove all locks with an **UNLOCK** statement before closing a file or terminating your program. Otherwise, undefined results will occur.

> **LOCK** and **UNLOCK** must match exactly.

■ Example 1

The following locks the entire file opened as number 2:

```
LOCK #2
```

The following locks only record 32 in file number 2:

```
LOCK #2, 32
```

The following locks records 1 through 32 in file number 2:

```
LOCK #2, TO 32
```

The following locks records 9 through 32 in file number 2:

```
LOCK #2, 9 to 32
```

■ Example 2

The following **UNLOCK** would be legal:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 4
UNLOCK #1, 5 TO 8
```

The following **UNLOCK** would be illegal, since the range in an
UNLOCK statement must match the range in the corresponding **LOCK**
statement exactly:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 8
```

■ Example 3

The following fragment opens a file. It then allows an operator to lock an
individual record and update the information in that record. When the
operator is done, the program unlocks the locked record. (Unlocking the
locked records allows other people to work with the file.)

```
100 OPEN "MONITOR" AS #1 LEN = 59
110 FIELD #1,15 AS PAYER$,20 AS ADDRESS$,20 AS PLACE$,4 AS OWE$
120 UPDATE$ = "Y"
```

LOCK...UNLOCK Statements

```
130 WHILE (UPDATE$ = "Y" OR UPDATE$ = "Y")
140    CLS:LOCATE 10,10
150    INPUT "CUSTOMER NUMBER?     #"; NUMBER%
160    LOCK #1, NUMBER%
170       GET #1, NUMBER%
180       DOLLARS! = CVS(OWE$)
190       LOCATE 11,10: PRINT "CUSTOMER: ";PAYER$
200       LOCATE 12,10: PRINT "ADDRESS:    ";ADDRESS$
210       LOCATE 13,10: PRINT "CURRENTLY OWES: $";DOLLARS!
220       LOCATE 15,10: INPUT "CHANGE (+ OR -)", CHANGE!
230       DOLLARS! = DOLLARS! + CHANGE!
240       LSET OWE$ = MKS$(DOLLARS!)
250       PUT #1, NUMBER%
260    UNLOCK #1, NUMBER%
270    LOCATE 17,10: INPUT "UPDATE ANOTHER? ", CONTINUE$
280    UPDATE$ = LEFT$(CONTINUE$,1)
290 WEND
```

■ **Syntax**

LOF(*filenumber*)

■ **Action**

Returns the length of the named file in bytes

■ **Remark**

When a file is opened for **APPEND** or **OUTPUT, LOF** returns the size of the file in bytes.

■ **Example**

In this example, the variables **REC** and **RECSIZ** contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

```
10IF REC*RECSIZ>LOF (1)
THEN PRINT "INVALID ENTRY"
```

LOG Function

■ **Syntax**

LOG(x)

■ **Action**

Returns the natural logarithm of x

■ **Remarks**

The x argument must be greater than zero.

The natural logarithm is the logarithm to the base e. The number e is approximately equal to 2.718282.

■ **Example 1**

```
PRINT LOG(45/7)
```

Output:

```
 1.860752
```

■ **Example 2**

```
100 E = 2.718282
110 FOR I = 1 TO 3
120 PRINT LOG(E^I)
130 NEXT I
```

Output:

```
1        2        3
```

■ **Syntax**

LPOS(*z*)

■ **Action**

Returns the current position of the printer's print head within the printer buffer

■ **Remarks**

The argument *x* is the index of the printer being tested. For example, **LPT1:** would be tested with LPOS (1), **LPT2:** with LPOS (2), and so on.

LPOS does not necessarily give the physical position of the print head.

■ **Example**

```
IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

■ **Syntax**

LPRINT [[*expressionlist*]] [[;]]
LPRINT USING *x$;expressionlist*

■ **Action**

Prints data on the printer

■ **Remarks**

These statements function the same as **PRINT** and **PRINT USING**, except output goes to the line printer, and the file number option is not permitted.

LPRINT assumes a 132-character-wide printer.

■ **Examples**

See "PRINT and PRINT USING Statements."

■ **Syntax**

LSET *stringvariable*=*x$*
RSET *stringvariable*=*x$*

■ **Action**

Moves data from memory to a random file buffer (in preparation for a **PUT** statement) or left- or right-justifies the value of a string in a string variable

■ **Remarks**

If *x$* requires fewer bytes than were fielded to *stringvariable,* **LSET** left-justifies the string in the field, and **RSET** right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are **LSET** or **RSET**.

Note

You may also use **LSET** or **RSET** with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines:

```
100 A$=SPACE$(20)
200 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

LSET and RSET Statements

■   **Example**

Line 150 converts the single-precision numeric variable AMT to a 4-byte string and stores that string in A$, left-justified. Line 160 converts the integer numeric variable to a 2-byte string and stores that string in D$, right-justified.

```
150 LSET A$=MKS$(AMT)
160 RSET D$=MKI$(COUNT%)
```

■ **Syntax**

MERGE *filespec*

■ **Action**

Merges a specified file into the program currently in memory.

■ **Remarks**

For merging a program not in memory, the *filespec* is an optional device specification followed by a file name or path name that conforms to operating system naming conventions for file names. BASIC appends the default file name extension **.BAS** if the user specifies no extensions, and the file has been saved to the disk.

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (Merging may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing a **MERGE** command.

■ **Example**

Inserts, by sequential line number, all lines in the program NUMBRS.BAS into the program currently in memory:

```
MERGE "NUMBRS"
```

■ **Compiler Differences**

The **MERGE** command is not supported by the compiler.

MID$ Function

■ **Syntax**

MID$($x$$,n[,m]))

■ **Action**

Returns a string of length m characters from $x$$, beginning with the nth character.

■ **Remarks**

The n and m must be in the range 1 to 255. If m is omitted or if there are fewer than m characters to the right of the nth character, all rightmost characters beginning with the nth character are returned. If n is greater than the number of characters in $x$$, MID$ returns a null string. (Use the LEN function to find the number of characters in $x$$.)

■ **Example**

```
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
```

Output:

```
GOOD EVENING
```

■ **See Also**

LEFT$, RIGHT$

214

■ **Syntax**

MID$$(x1\$, n[, m]) = x2\$$

■ **Action**

Replaces a portion of one string with another string

■ **Remarks**

The n and m are integer expressions and $x1\$$ and $x2\$$ are string expressions.

The characters in $x1\$$, beginning at position n, are replaced by the characters in $x2\$$. The optional m refers to the number of characters from $x2\$$ that are used in the replacement. If m is omitted, all of string $x2$ is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of $x1\$$.

■ **Example**

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
```

Output:

```
KANSAS CITY, KS
```

MID$ is also a *function* that returns a substring of a given string.

■ **See Also**

MID$ Function

MKDIR Statement

■ **Syntax**

MKDIR *pathname*

■ **Action**

Creates a new directory

■ **Remarks**

The *pathname* is a string expression specifying the name of the directory which is to be created. **MKDIR** works exactly like the operating system command **MKDIR**. The *pathname* must be a string of less than 128 characters.

■ **Example**

Assume the current directory is the root. The following creates a subdirectory named SALES in the current directory of the current drive:

```
MKDIR "SALES"
```

The following example creates a subdirectory named USERS in the current directory of drive B:

```
MKDIR "B:USERS
```

■ **See Also**

CHDIR, RMDIR

■ **Syntax**

MKI$ (*integer-expression*)
MKS$ (*single-precision-expression*)
MKD$ (*double-precision-expression*)

■ **Action**

Converts numeric values to string values

■ **Remarks**

Any numeric value that is placed in a random file buffer with an **LSET** or **RSET** statement must be converted to a string. **MKI$** converts an integer to a 2-byte string. **MKS$** converts a single precision number to a 4-byte string. **MKD$** converts a double precision number to an 8-byte string.

These functions differ from **STR$** in that they do not really change the data bytes, just the way BASIC interprets those bytes.

■ **Example**

Line 110 of this example converts the single precision value in the variable AMT to a 2-byte string which is part of **FIELD**.

```
90   AMT=(K+T)
100  FIELD #1,8 AS D$,20 AS N$
110  LSET D$=MKS$(AMT)
120  LSET N$=A$
130  PUT #1
 .
 .
 .
```

■ **See Also**

CVI, CVS, CVD Functions

MOTOR Statement

MOTOR *state*

■ **Action**

Turns the cassette motor on or off.

■ **Remarks**

The *state* argument is a Boolean value indicating on or off. If *state* is non-zero, the cassette motor is turned on. If the state is zero, the motor is turned off.

If the *state* is omitted, **MOTOR** works as a toggle switch. That is, if the motor is off, it is turned on, and vice versa.

■ **Example**

```
10 MOTOR 1    'Turn motor on
20 MOTOR 0    'Turn motor off
30 MOTOR      'Turn motor back on.
```

■ **Compiler/Interpreter Differences**

The **MOTOR** statement is not supported by the compiler.

■ **Syntax**

NAME *oldfilename* AS *newfilename*

■ **Action**

Change the name of a disk file

■ **Remarks**

The *oldfilename* must exist and the *newfilename* must not exist; otherwise, an error will result. Also, both files must be on the same drive.

You may not rename a file with a new drive designation. If you attempt this, a "Rename across disks" error will be generated. After a NAME command, the file exists on the same disk, in the same disk space, but with the new name.

NAME has much the same effect as copying and then deleting a file, except that, with NAME, the file is not moved, only its directory entry.

You may not use NAME to rename directories.

You must be close the file you wish to rename before using NAME. Also, there must be one free file handle.

■ **Examples**

In this example, the file that was formerly named ACCTS is now named LEDGER:

NAME "ACCTS" AS "LEDGER"

NAME may be used to move a file from one directory to another. For example:

NAME "\X\CLIENTS" AS "\XYZ\P\CLIENTS"

NEW Command

■ **Syntax**

NEW

■ **Action**

Deletes the program currently in memory and clears all variables

■ **Remarks**

NEW is entered in direct mode to clear memory before entering a new pro-gram. BASIC always returns to command level after a NEW is executed.

NEW closes all files and turns tracing off.

■ **Example**

NEW

■ **Compiler Differences**

The NEW command is not supported by the compiler.

■ **Syntax**

OCT$(*expression*)

■ **Action**

Returns a string that represents the octal value of the decimal argument x. The argument x is rounded to an integer before OCT$(*expression*) is evaluated.

■ **Example**

```
PRINT OCT$(24)
```

Output:

```
30
```

■ **See Also**

HEX$

ON COM Statement

■ Syntax

ON COM(*n*) GOSUB *linenumber*

■ Action

Specifies the first line number of a subroutine to be performed when activity occurs on a communications port

■ Remarks

The *n* argument is the number of the communications port.

The *linenumber* is the number of the first line of a subroutine that is to be performed when activity occurs on the specified communications port.

A *linenumber* of zero disables the communications event trap.

The ON COM statement will only be executed if a COM(*n*) ON statement has been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the ON COM statement is not zero, GW-BASIC checks between statements to see if communications activity has occurred on the specified port. If communications activity has occurred, a GOSUB will be performed to the specified line.

If a COM OFF statement has been executed for the communications port, the GOSUB is not performed and is not remembered.

If a COM STOP statement has been executed for the communications port, the GOSUB is not performed, but will be performed as soon as a COM ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic COM STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a COM ON statement unless an explicit COM OFF was performed inside the subroutine.

The RETURN *linenumber* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

222

Event trapping does not take place when GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

■ **Compiler/Interpreter Differences**

With the compiler, you must use the /V or /W option on the compiler command line if a program contains an ON COM statement. These options allow the compiler to function correctly when event trapping routines are included in a program.

ON ERROR GOTO Statement

■ Syntax

ON ERROR GOTO *linenumber*

■ Action

Enables error handling and specifies the first line of the error handling routine

■ Remarks

Once error handling has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error handling routine. If *linenumber* does not exist, an "Undefined line" error results.

To disable error handling, execute the following statement:

```
ON ERROR GOTO 0.
```

Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes Microsoft GW-BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

■ **Example**

```
10 ON ERROR GOTO 1000
```

■ **Compiler/Interpreter Differences**

With the compiler, you must use the /E compilation option on the compiler command line if a program contains **ON ERROR GOTO** and **RESUME** *linenumber* statements. If the **RESUME, RESUME NEXT**, or **RESUME 0** form is used, then specify the /X option instead.

If you do not use the /X option, at least one integer line number must appear in the program before the statement that causes the error.

Regardless of the switch you use (/E or /X), the compiler supports the use of line labels as the objects of **RESUME** and **ON ERROR GOTO** statements.

The purpose of these options is to allow the compiler to function correctly when error handling routines are included in a program. Note that the use of these options increases the size of OBJ and EXE files.

ON...GOSUB and ON...GOTO Statements

■ **Syntax**

ON *expression* **GOTO** *linenumber-list*
ON *expression* **GOSUB** *linenumber-list*

■ **Action**

Branches to one of several specified line numbers, depending on the value returned when an expression is evaluated

■ **Remarks**

The value of *expression* determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is not an integer, the number is rounded.)

In the **ON...GOSUB** statement, each line number in the list must be the first line number of a subroutine.

If the value of *expression* is either zero or greater than the number of items in the list, control drops to the next BASIC statement.

If the value of *expression* is negative or greater than 255, an "Illegal function call" error occurs.

■ **Example**

In this example, if the value of L were 4, then control would branch to statement 320:

```
100 ON L-1 GOTO 150,300,320,390
```

■ **Compiler/Interpreter Differences**

The compiler does not check the value of *expression,* except to make sure that it does not exceed the number of items in the *linenumber-list.* If it does exceed that number, then it will produce erroneous results.

■ **Syntax**

ON KEY(*n*) GOSUB *linenumber*

■ **Action**

Specifies the first line number of a subroutine to be performed when the given key is pressed

■ **Remarks**

The argument *n* is the number of a function key, direction key, or user-defined key.

The *linenumber* is the number of the first line of a subroutine that is performed when you press the specified function or cursor direction key. A *linenumber* of zero disables the event trap.

The **ON KEY** statement is executed only if a **KEY**(*n*) **ON** statement has been executed to enable event trapping. If key trapping is enabled, and if the *linenumber* in the **ON KEY** statement is not zero, GW-BASIC checks between statements to see if you have pressed the specified function, user-defined or cursor direction key. If you have, the program branches to a subroutine specified by the **GOSUB** statement.

If the key has been turned off with a **KEY**(*n*) **OFF** statement, the **GOSUB** is not performed and is not remembered.

If a **KEY STOP** statement has been executed for the specified key, the **GOSUB** is not performed, but will be performed as soon as a **KEY**(*n*) **ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **KEY**(*n*) **STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine automatically performs a **KEY**(*n*) **ON** statement unless an explicit **KEY**(*n*) **OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

ON KEY Statement

Event trapping does not take place when GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

The following rules apply to keys trapped by BASIC:

1. The line printer echo toggle key is processed first. Defining this key as a user-defined key trap will not prevent characters from being echoed to the line printer if depressed.
2. Function keys and the cursor direction keys are examined next. Defining a function key or cursor direction key as a user-defined key trap will have no effect as they are considered pre-defined.
3. Finally, the user-defined keys are examined.
4. Any key that is trapped is not passed on. That is, the key is not read by BASIC.

Warning

This may apply to any key, including Break or system reset (warm boot)! This is a powerful feature when you consider that it is now possible to prevent accidentally breaking out of a program, or worse yet, rebooting the machine.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the **INPUT** or **INKEY**$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

The **ON KEY**([*n*]) statement allows six additional user-defined **KEY** traps. This allows any key, control-key, shift-key, or super-shift-key to be trapped by the user as follows:

ON KEY(*n*) **GOSUB** *linenumber*

Where: *n* is an integer expressing a legal user-defined key number.

■ **Examples**

In the following, the programmer has overridden the normal function asso-
ciated with function key 4, and replaced it with

SCREEN 0,0

which is printed whenever that key is pressed. The value may be reassigned
and it will resume its standard function when the machine is rebooted.

```
10 KEY 4,"SCREEN 0,0"   'assigns softkey 4
20 KEY(4) ON            'enables event trapping
.
.
70 ON KEY(4) GOSUB 200
.
.   key 4 pressed
.
200   'Subroutine for screen
```

In the following, the programmer has enabled the CONTROL-S key to enter a
subroutine which closes the files and stops program execution until he or
she is ready to continue.

```
100 KEY 15, CHR$(&H04) + CHR$(83)
105 REM Key 15 now is Control-S
110 KEY(15) ON
.
.
.
1000 PRINT "If you want to stop processing for a
           break"
1010 PRINT "press the Control key and the 'S' at
           the"
1020 PRINT "same time."
1030 ON KEY(15) GOSUB 3000.
.
.   Operator presses Control-S
.
3000 REM ** Suspend processing loop.
3010 CLOSE 1
3020 RESET
3030 CLS
3035 PRINT "Enter CONT to continue."
3040 STOP
3050 OPEN "A", 1, "ACCOUNTS.DAT"
3060 RETURN
```

ON KEY Statement

■   **Compiler/Interpreter Differences**

With the compiler, you must use the /V or /W option on the compiler
command line if a program contains an ON KEY statement. These
options allow the compiler to function correctly when you include event-
trapping routines in a program.

■ **Syntax**

ON PEN GOSUB *linenumber*

■ **Action**

Specifies the first line number of a subroutine to be performed when the lightpen is activated

■ **Remarks**

The *linenumber* is the number of the first line of a subroutine that is performed when you activate the lightpen. A *linenumber* of zero disables the pen event trap.

The ON PEN statement will only be executed if a PEN ON statement has been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the ON PEN statement is not zero, GW-BASIC checks between statements to see if the lightpen has been activated. When you activate the lightpen, a GOSUB is performed to the specified line.

If a PEN OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a PEN STOP statement has been executed, the GOSUB is not performed, but will be performed as soon as a PEN ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic PEN STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a PEN ON statement unless an explicit PEN OFF was performed inside the subroutine.

You may use the RETURN *linenumber* form of the RETURN statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

ON PEN Statement

Event trapping takes place only when GW-BASIC is executing a program; event trapping is automatically disabled when an error trap occurs.

■ **Compiler/Interpreter Differences**

With the compiler, you must use the /V or /W option in the compiler command line if a program contains an ON PEN statement. These switches allow the compiler to function correctly when you include event trapping routines in a program.

■ **Syntax**

ON PLAY (*queuelimit*) GOSUB *linenumber*

■ **Action**

Branches to a specified subroutine when the music queue contains fewer than *queuelimit* notes. This permits continuous music during program execution.

■ **Remarks**

The *queuelimit* argument is an integer expression in the range 1 through 32. Values outside this range result in an "Illegal function call" error.

The *linenumber* is the statement line number of the **PLAY** ev ent trap subroutine. **ON PLAY** causes an event trap when the background music queue goes from *queuelimit* notes to *queuelimit*-1 notes.

The **ON PLAY** statement is used with the **PLAY ON, PLAY OFF**, and **PLAY STOP** statements which enable, disable, and suspend **PLAY** event trapping, respectively.

If **PLAY ON** is executed, then a **GOSUB** is executed when a **PLAY** even occurs.

If **PLAY OFF** is executed, then no **GOSUB** is executed and the statement is not remembered.

If a **PLAY STOP** statement has been executed, the **GOSUB** is not performed, but will be performed as soon as a **PLAYON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **PLAY STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine automatically performs a **PLAY ON** statement unless an explicit **PLAY OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB, WHILE,** or **FOR** statement that is active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result. Rules:

ON PLAY Statement

1. A play event trap occurs only when playing background music Play event traps do not occur when running in the music foreground.

2. A play event trap does not occur if the background music queue has already gone from having n to $n-1$ notes when a **PLAY ON** is executed.

3. If n is a large number, event traps will occur frequently enough to diminish program execution speed.

■ **See Also**

PLAY ON, PLAY OFF, PLAY STOP

■ **Example**

In this example, control branches to a subroutine when the background music buffer decreases to 7 notes.

```
100 PLAY ON
.
.
540 PLAY "MB L1 XZITHER$"
550 ON PLAY(8) GOSUB 6000
.
.
.
6000 REM **BACKGROUND MUSIC**
6010 LET COUNT% = COUNT% + 1
.
.
6999 RETURN
```

■ **Syntax**

ON STRIG(*n*) GOSUB *linenumber*

■ **Action**

Specifies the first line number of a subroutine to be performed when the
joystick trigger is pressed

■ **Remarks**

The argument *n* is the number of the joystick trigger.

The *linenumber* is the number of the first line of a subroutine that is per-
formed when you press the joystick trigger. A *linenumber* of zero disables
the event trap.

The ON STRIG statement is executed only if a STRIG ON statement
has been executed to enable event trapping. If event trapping is enabled,
and if the *linenumber* in the ON STRIG statement is not zero, GW-BASIC
checks between statements to see if you have pressed the joystick trigger.
If you have, a GOSUB is performed to the specified line.

If a STRIG OFF statement has been executed, the GOSUB is not per-
formed and is not remembered.

If a STRIG STOP statement has been executed, the GOSUB is not per-
formed, but will be performed as soon as a STRIG ON statement is exe-
cuted.

When an event trap occurs (i.e., the GOSUB is performed), an automatic
STRIG STOP is executed so that recursive traps cannot take place. The
RETURN from the trapping subroutine will automatically perform a
STRIG ON statement unless an explicit STRIG OFF was performed
inside the subroutine.

You may use the RETURN *linenumber* form of the RETURN statement
to return to a specific line number from the trapping subroutine. Use this
type of return with care, however, because any other GOSUB, WHILE, or
FOR statements that were active at the time of the trap will remain
active, and errors such as "FOR without NEXT" may result.

ON STRIG Statement

Event trapping does not take place when GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

■ **Compiler/Interpreter Differences**

With the compiler, you must use the /V or /W option on the compiler command line if a program contains an **ON STRIG** statement. These switches allow the compiler to function correctly when you include event trapping routines in a program.

■ **Syntax**

ON TIMER (*n*) GOSUB *linenumber*

■ **Action**

Provides an event trap during real time

■ **Remarks**

ON TIMER causes an event trap every *n* seconds. The *n* argument must be a numeric expression in the range of 1 to 86400 (1 second to 24 hours). Values outside this range generate an "Illegal function call" error.

The **ON TIMER** statement is executed only if a **TIMER ON** statement has been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the **ON TIMER** statement is not zero, GW-BASIC checks between statements to see if the time has been reached. If it has, a **GOSUB** is performed to the specified line.

If a **TIMER OFF** statement has been executed the **GOSUB** is not performed and is not remembered.

If a **TIMER STOP** statement has been executed, the is not performed, but will be performed as soon as a **TIMER ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **TIMER STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine will automatically perform a **TIMER ON** statement unless an explicit **TIMER OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

ON TIMER Statement

■ Example

This example displays the time of day on line 1 every minute:

```
20 TIMER ON
.
.
.
10000 OLDROW=CSRLIN 'Save current Row
10010 OLDCOL=POS(0)  'Save current Column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'Restore Row & Col
10040 RETURN
```

■ See Also

TIMER ON, TIMER OFF, TIMER STOP

■ **Syntax**

OPEN *"filespec"* [FOR *mode1*][ACCESS *atype*] [*ltype*] AS [#] *filenum* [LEN=*reclen*

OPEN *mode2*,[#]*filenum, filespec* [,*reclen*] obsolescent

■ **Action**

Allows I/O to a file or device

■ **Remarks**

The *filespec* is an optional device specification followed by a file name or path name that conforms to DOS file naming conventions.

In the first syntax, *mode1* is an expression that is one of the following:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append to) the file.
RANDOM	Specifies random file mode, which is the default file mode. In RANDOM mode, if no ACCESS clause is present, three attempts are made to open the file when the OPEN statement is executed. Access is attempted in the following order:

1. Read/write

2. Write-only

3. Read-only

ACCESS *atype*
> Specifies the type of operation to be performed on the opened file. If the file is already opened by another process and access of the type specified is not allowed, the OPEN will fail and a "Permission Denied" error message is generated.
>
> The *atype* may be one of the following:

READ	Opens the file for reading only.
WRITE	Opens the file for writing only.
READ WRITE	Opens the file for both reading and writing. This mode is valid only for **RANDOM** files and files opened for **APPEND**.

If *mode1* is omitted, the default random access mode is assumed. Random, however, cannot be expressed explicitly as the file mode.

The *ltype* clause works in a multiprocessing environment to restrict access by other processes to an open file. The *ltypes* are:

default	If *ltype* is not specified, the file may be opened for reading and writing any number of times by this process, but other processes are denied access to the file while it is opened.
SHARED	Any process on any machine may read from or write to this file. (Do not confuse this with the **SHARED** statement or the **SHARED** attribute to the **COMMON, DIM,** and **REDIM** statements.)
LOCK READ	No other process is granted read access to this file. This access is granted only if no other process has a previous **READ** access to the file.
LOCK WRITE	No other process is granted write access to this file. This lock is granted only if no other process has a previous **WRITE** access to the file.
LOCK READ WRITE	
	No other process is granted either read or write access to this file. This access is granted only if **READ** or **WRITE** access has not already been granted to another process, or a **LOCK READ** or **LOCK WRITE** is not already in place.

When the **OPEN** is restricted by a previous process, it generates error 70, "Permission Denied" under DOS

The *filenum* is an integer expression whose value is between 1 and 255. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

The *recordlen* is an integer expression that, if included, sets the record length for random files. BASIC ignores this option if it is used in a statement to OPEN a sequential file. The default length for records is 128 bytes, unless the command line options /I and /Rhavebeenused.

The *mode2* argument is a string expression. The first character must be one of the following:

O   Specifies sequential output mode.

I   Specifies sequential input mode.

R   Specifies random input/output mode.

A   Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A **PRINT#** or **WRITE#** statement will then extend (append to) the file.

You must open a file before any I/O operation can be performed on that file. **OPEN** allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

OPEN allows *pathname* in place of *filespec*. If you use a pathname, and include a drive in the pathname, you must include the drive at the beginning of the pathname. That is, "B:\SALES\JOHN" is legal, while "\SALES\B:JOHN" is *not* legal.

The **LEN=** option is ignored if the file being opened has been specified as a sequential file.

Since it is possible to refer to the same file in a subdirectory via different paths, it is nearly impossible for BASIC to know that it is the same file simply by looking at the path. For this reason, BASIC will not let you open the file for **OUTPUT** or **APPEND** if it is on the same disk even if the path is different. For example, if MARY is your current directory, then:

```
OPEN    "REPORT"     ...
OPEN    "\SALES\MARY\REPORT"...
OPEN    "..\MARY\REPORT"   ...
OPEN    "..\..MARY\REPORT" ...
```

all refer to the same file.

OPEN Statement

■ DOS Devices

BASIC devices are:

KYBD:	LPT*n*:
SCRN:	CON:
COM*n*:	CAS*n*:

The BASIC file I/O system allows you to take advantage of user-installed devices. (See your DOS manual for information on character devices.)

Opened character devices are opened and used in the same manner as disk files. However, characters are not buffered by BASIC as they are for disk files. The record length is set to one.

BASIC only sends a CR (carriage return X'0D') at the end of line. If the device requires a LF (line feed X'0A'), the driver must provide it. When writing device drivers, keep in mind that other BASIC users will want to read and write control information. Writing and reading of device control data is handled by the BASIC **IOCTL** statement and **IOCTL$( )** function.

Note

A file can be opened for sequential input or random access on more than one file number at a time. A file may be opened for output, however, on only one file number at a time.

■ Examples

The following example opens MAILING.DAT as file number 1, and allows data to be added without destroying what is already in MAILING.DAT:

OPEN "MAILING.DAT" FOR APPEND AS 1

If you wrote and installed a device named ROBOT, then the **OPEN** statement might appear as:

OPEN "\DEV\ROBOT" FOR OUTPUT AS 1

To open the printer for output, you could use the line:

```
OPEN "LPT:" FOR OUTPUT AS 1
```

which uses the BASIC device driver, or as part of a pathname as in:

```
OPEN "\DEV\LPT1" FOR OUTPUT AS 1
```

which uses the DOS device driver.

The following statement opens the file RECORDS in random mode, for reading only. The statement locks the file for writing, but allows reading by other processes while the OPEN is in effect:

```
OPEN "RECORDS" FOR RANDOM ACCESS READ LOCK WRITE AS #1
.
.
.
```

The following example opens the file named INVEN for input as file number 2:

```
OPEN "I", 2, "INVEN"
```

OPEN COM Statement

■ **Syntax**

OPEN "COM*n*: [[*speed*]][, [*parity*]] [, [*data*]][, [*stop*]]
    [,RS] [,CS[[*n*]]] [,DS[[*n*]]]
    [,CD[[*n*]]] [,BIN] [,ASC] [,LF]]]]]"
    [FOR*mode* AS [#]*filenumber* [LEN= *reclen*]]

■ **Action**

Opens and initializes a communications channel for input/output

■ **Remarks**

COM*n*: is the name of the device to be opened.

The *n* argument is the number of a legal communications device, i.e.,
COM1: or **COM2:**.

The *speed* is the baud rate, in bits per second, of the device to be opened.

The *parity* designates the parity of the device to be opened. Valid entries
are: N (none), E (even), O (odd), S (space), or M (mark).

The *data* argument designates the number of data bits per byte. Valid
entries are: 5, 6, 7, or 8.

The *stop* argument designates the stop bit. Valid entriesare: 1, 1.5, or 2.

RS suppresses RTS (Request To Send).

CS[*n*] controls CTS (Clear To Send).

DS[*n*] controls DSR (Data Set Ready).

CD[*n*] controls CD (Carrier Detect).

LF allows communication files to be printed on a serial line printer. When
LF is specified, a linefeed character (0AH) is automatically sent after each
carriage return character (0CH). This includes the carriage return sent as a
result of the width setting. Note that **INPUT** and **LINE INPUT**, when
used to read from a **COM** file that was opened with the **LF** option, stop
when they see a carriage return, ignoring the linefeed.

The **LF** option is superseded by the **BIN** option.

BIN opens the device in binary mode. **BIN** is selected by default unless **ASC** is specified.

In the **BIN** mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and CONTROL-Z is not treated as end-of-file. When the channel is closed, CONTROL-Z will not be sent over the RS232 line. The **BIN** option supersedes the **LF** option.

ASC opens the device in ASCII mode. In ASCII mode, tabs are expanded, carriage returns are forced at the end-of-line, CONTROL-Z is treated as end-of-file, and the XON/XOFF protocol is enabled. When the channel is closed, CONTROL-Z is sent over the RS-232 line.

The *mode* argument is one of the following string expressions:

OUTPUT Specifies sequential output mode.

INPUT Specifies sequential input mode.

If the *mode* expression is omitted, it is assumed to be random input/output. Random input/output cannot, however, be explicitly chosen as *mode*.

The *filenumber* is the number of the file to be opened. The **OPEN COM** statement must be executed before a device can be used for RS-232 communication. Any syntax errors in the **OPEN COM** statement will result in a "Bad File name" error.

The *speed, parity, data,* and *stop* options must be listed in the order shown in the above syntax. The remaining options may be listed in any order, but they must be listed after the *speed, parity, data,* and *stop* options.

A "Device timeout" error occurs if Data Set Ready (DSR) is not detected.

■ **Example**

The following opens communications channel 1 in random mode at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/Output will be in the binary mode. Other lines in the program may now access channel 1 as file number 2.

```
10 OPEN "COM1:9600,N,8,1,BIN" AS 2
```

OPTION BASE Statement

■ **Syntax**

OPTION BASE n

■ **Action**

Declares the minimum value for array subscripts.

■ **Remarks**

The value of n must be either 0 or 1. The default base is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is 1.

You must include the **OPTION BASE** statement *before* you define or use any arrays. Repeated **OPTION BASE** statements are permitted, but they must all be identical or a "Duplicate Definition" error results.

Chained programs may have an **OPTION BASE** statement if no arrays are passed between them or the specified base is identical in the chained programs. The chained program will inherit the **OPTION BASE** value of the chaining program.

■ **Example**

The following statement overrides the default value of zero, so the lowest value a subscript in an array may have in this program is 1:

10 OPTION BASE 1

■ **Compiler/Interpreter Differences**

The compiler does not "execute" an **OPTION BASE** statement, as it does a **PRINT** statement, for example. An **OPTION BASE** statement takes effect as soon as it is encountered in the program during compilation. This option base then remains in effect until the end of the program or until another **OPTION BASE** statement is encountered in program.

■ Syntax

OUT *port, data*

■ Action

Sends a byte to a machine output port

■ Remarks

The *port* is the port number. It must be an integer expression in the range 0 to 65535.

The *data* argument is the data to be transmitted. It must be an integer expression in the range 0 to 255.

■ Example

```
100 OUT 12345,255
```

In 8086 assembly language, this is equivalent to:

```
MOV DX,12345
MOV AL,255
OUT DX,AL
```

PAINT Statement

■ **Syntax**

PAINT (x,y) [[,*paint* [[,*bordercolor*] [[,*background*]]]]

■ **Action**

Fills a graphics area with the color or pattern specified

■ **Remarks**

The coordinates show the point where painting begins. This point should be either inside or outside the figure you want to paint; never on the border itself. If this point is on the inside, then naturally enough the interior of the figure is painted; if the point is on the outside, the background is painted.

If the *paint* argument is a string expression, then **PAINT** executes "tiling," a process similar to "line-styling." Like **LINE**, **PAINT** looks at a "tiling" mask each time a point is put down on the screen.

If *paint* is a numeric expression, then the number must be a valid color. It is used to paint the area as before. If the you do not specify *paint,* the foreground color is used.

The *bordercolor* identifies the border color of the figure you are painting. When the border color is encountered, painting of the current line stops. If the *bordercolor* is not specified, the *paint* argument is used.

The *background* is a string formula returning character data. When it is omitted, the default is CHR$ (O).

When specified, *background* gives the "background tile slice" to skip when checking for termination of the boundary. Painting is terminated when adjacent points display the paint color; however; specifying a background tile slice allows you to paint over an already painted area.

Painting is complete when a line is painted without changing the color of any pixel; in other words, when the entire line is equal to the paint color.

You can use the **PAINT** command to fill any figure, but painting complex figures may result in an "Out of Memory" error. If this happens, you may want to use the **CLEAR** statement at the beginning of your program to increase the amount of stack space available.

The **PAINT** command permits coordinates outside the screen or viewport.

■ **Tiling**

Tiling is the design of a **PAINT** pattern that is 8 bits wide and up to 64 bytes long. Each byte in the tile string masks 8 bits along the x axis when putting down points. Construction of this tile mask works as follows:

Use the syntax

PAINT (x,y), CHR\$$(n)$...CHR\$$(n>)$

where n is a number between 0 and 255 which is represented in binary across the x-axis of the tile. Each **CHR\$$(n)$** up to 64 generates an image not of the assigned character, but of the bit arrangement of the code for that character. For example, the decimal number 85 is binary "01010101"; the graphic image line on a black and white screen generated by CHR\$(85) is an eight pixel line, with even numbered points turned white, and odd ones black. That is, each bit containing a 1 sets the associated pixel "on" and each bit filled with a 0 sets the associated bit "off" in a black and white system. The ASCII character CHR\$(85), which is U, is not displayed in this case.

If the current screen mode supports only two colors, then the screen can be painted with X images with the following statement:

```
10 SCREEN 2
20 A$=CHR$ (129) +CHR$ (66) +CHR$ (36) +CHR$ (24)
30 B$=CHR$ (24) +CHR$ (36) +CHR$ (66) +CHR$ (129)
40 PAINT (320,100) ,A$+B$
```

This appears on the screen as:

```
      x increases -->
0,0|x| | | | | | |x|  CHR$(129)   Tile byte 1
0,1| |x| | | | |x| |  CHR$(66)    Tile byte 2
0,2| | |x| | |x| | |  CHR$(36)    Tile byte 3
0,3| | | |x|x| | | |  CHR$(24)    Tile byte 4
0,4| | | |x|x| | | |  CHR$(24)    Tile byte 5
0,5| | |x| | |x| | |  CHR$(36)    Tile byte 6
0,6| |x| | | | |x| |  CHR$(66)    Tile byte 7
0,7|x| | | | | | |x|  CHR$(129)   Tile byte 8
```

PAINT Statement

When supplied, *background* specifies the "background tile slice" to skip when checking for boundary termination.

You cannot specify more than two consecutive bytes that match the tile string in the tile background slice. Specifying more than two will result in an "Illegal function call" error.

■ Example

The following begins painting at coordinates 5,15 with color 2 and border color 0, and fills to a border:

```
PAINT (5,15),2,0
```

■ **Syntax**

PALETTE [[*attribute,color*]]
PALETTE USING *arrayname (arrayindex)*

■ **Action**

Changes one or more of the colors in the palette

■ **Remarks**

The **PALETTE** statement works only for systems equipped with the Enhanced Graphics Adapter. A BASIC palette contains a set of colors, with each color specified by an *attribute*. Each *attribute* is paired with an actual display *color*. This *color* determines the actual visual color on the screen, and is dependent on the setting of your screen mode and your actual physical hardware display. The **PALETTE** statement works *only* for systems with the IBM Extended Graphics Adapter (EGA).

PALETTE with no arguments sets the palette to a known initial setting. This setting is the same as the setting when colors are first initialized.

If arguments are specified, *color* will be displayed whenever *attribute* is specified in any statement that specifies a color. Any color changes on the screen occur immediately. Note that when graphics statements use color arguments, they are actually referring to attributes and not actual colors. **PALETTE** pairs attributes with actual colors.

For example, assume that the current palette consists of *colors* 0, 1, 2, and 3. The following **DRAW** statement

DRAW "C3L100"

selects attribute 3, and draws a line of 100 pixels using the color associated with the attribute 3, in this case, also 3. If the statement

PALETTE 3,2

is executed, then the color associated with attribute 3 is changed to color 2. All text or graphics currently on the screen displayed using attribute 3 are instantaneously changed to color 2. All text or graphics subsequently displayed with attribute 3 will also be displayed in color 2. The new palette of *colors* will contain 0, 1, 2, and 2.

PALETTE, PALETTE USING Statements

With the USING option, all entries in the palette can be modified in one
PALETTE statement. The *arrayname* argument is the name of an integer
array and the *arrayindex* specifies the index of the first array element in the
arrayname to use in setting your palette. Each *attribute* in the palette is
consecutively assigned to the the respective *color* in the array. If the *color*
argument in an array entry is -1, then the mapping for the associated *attri-
bute* is not changed. All other negative numbers are illegal values for *color*.

You can use the color argument in the COLOR statement to set the
default text color. (Remember that color arguments in other BASIC state-
ments are actually what are called *attributes* in this discussion.) This color
argument specifies the way that text characters appear on the display
screen. Under a common initial palette setting, points colored with the
attribute 0 appear as black on the display screen. Using the PALETTE
statement, you could, for example, change the mapping of *attribute* 0 from
black to white.

Remember, that a PALETTE statement executed without any parameters
will assign all *attributes* their default *colors*.

The following table lists *attribute* and *color* ranges for various monitor
types and screen modes.

Table 6.1

SCREEN Color and Attribute Ranges

SCREEN Mode	Monitor Attached	Adapter	Attribute Range	Color Range
0	Monochrome	MDPA	NA	NA
	Monochrome	EGA	0-15	0-2
	Color	CGA	NA	0-15
	Color/Enhanced	EGA	0-15	0-15
1	Color	CGA	NA	0-3
	Color/Enhanced	EGA	0-3	0-15
2	Color	CGA	NA	0-1
	Color/Enhanced	EGA	0-1	0-15
7	Color/Enhanced	EGA	0-15	0-15
8	Color/Enhanced	EGA	0-15	0-15
9	Enhanced	EGA	0-3	0-63
	Enhanced	EGA+	0-15	0-63
10	Monochrome	EGA	0-3	0-8

＊ Attributes 16-15 refer to blinking versions of colors 0-15
+ With greater than 64K of EGA memory
NA=Not Applicable.
CGA=IBM Color Graphics Adapter
EGA=IBM Extended Graphics Adapter
MDPA=IBM Monochrome Display and Printer Adapter

See the "SCREEN Statement" for the list of colors available for various **SCREEN** mode, monitor, and graphics adapter combinations.

■ **Example**

The following changes all points colored with *attribute* 0 to *color* 2:

```
PALETTE 0,2
```

This does not modify the palette:

```
PALETTE 0,-1
```

The following changes each palette entry. All *attributes* are now mapped to

display color zero (since the array is initialized to zero when it is first declared).

PALETTE A%(0)

The screen will appear as one single color. However, it will still be possible to execute any GW-BASIC statements.

The following example sets each palette entry to its appropriate initial display color:

PALETTE

Actual initial colors depend on your screen hardware configuration.

■ **Syntax**

PCOPY *sourcepage, destinationpage*

■ **Action**

Copies one screen page to another in all screen modes

■ **Remarks**

The *sourcepage* is an integer expression in the range 0 to n, where n is determined by the current video memory size and the size per page for the current screen mode.

The *destinationpage* has the same requirements as the *sourcepage*.

■ **See Also**

CLEAR, SCREEN

■ **Example**

This copies the contents of page 1 to page 2:

PCOPY 1, 2

PEEK Function

■ Syntax

PEEK(n)

■ Action

Returns the byte read from the indicated memory location n.

■ Remarks

The returned value is an integer in the range 0 to 255. The integer n must be in the range -32768 to 65535. The n argument is the offset from the current segment, which was defined by the last **DEF SEG** statement. For the interpretation of a negative value of n, see the "VARPTR Function."

PEEK is the complementary function of the **POKE** statement.

■ Example

A=PEEK (&H5A00)

In this example, the value at the location with the hexadecimal address 5A00 is loaded into the variable A.

■ **Syntax**

PEN(*n*)

■ **Remarks**

The argument *n* is a numeric expression returning an unsigned integer in the range 0 to 9 (see below).

The **PEN(*n*)** function reads the light pen coordinates, where *n* is:

0	If pen was down since last poll. Returns -1 if down, 0 if not.
1	Returns the x pixel coordinate where pen was last pressed.
2	Returns the y pixel coordinate where pen was last pressed.
3	Returns the current pen switch value: -1 if down, 0 if up.
4	Returns the last known valid x pixel coordinate.
5	Returns the last known valid y pixel coordinate.
6	Returns the character row position where pen was last pressed.
7	Returns the character column position where pen was last pressed.
8	Returns the last known character row where the pen was positioned.
9	Returns the last known valid character column where the pen was positioned.

■ **Example**

The following example produces an endless loop to print the current pen switch status (UP/DOWN):

```
5   CLS
10 PEN ON
20 P=PEN(3)
30 LOCATE 1,1 : PRINT "PEN IS"
40 IF P THEN PRINT "DOWN" ELSE PRINT "UP"
50 GOTO 20
```

PEN ON, PEN OFF, PEN STOP Statements

■ **Syntax**

PEN ON
PEN OFF
PEN STOP

■ **Action**

The **PEN ON** statement enables the lightpen read function and event trapping.

The **PEN OFF** statement disables the lightpen read function and event trapping.

The **PEN STOP** statement disables the lightpen read function and event trapping but remembers a **PEN** event so that it can be trapped as soon as event trapping is enabled.

■ **Remarks**

The pen is initially off. A **PEN ON** statement must be executed before any pen read function calls. If a read pen function is called when the pen is off, an "Illegal function call" error will result.

PEN ON also enables event trapping with an **ON PEN** statement.

To speed program execution, the pen should be turned off with a **PEN OFF**

■ **See Also**

PEN ON

■ **Example 1**

```
10 PEN ON 'enables lightpen read function and event trap
```

■ **Example 2**

```
20 PEN OFF 'disables lightpen read function and event trap
```

■ **Compiler/Interpreter Differences**

See compiler note under "ON PEN Statement."

PLAY Function

- **Syntax**

PLAY (*n*)

- **Action**

Returns the number of notes currently in the background music queue

- **Remarks**

The argument *n* is a dummy argument and may be any value.

PLAY(*n*) will return 0 when the user is in music foreground mode.

PLAY ON, PLAY OFF, PLAY STOP Statements

■ **Syntax**

PLAY ON
PLAY OFF
PLAY STOP

■ **Action**

PLAY ON enables play event trapping
PLAY OFF disables play event trapping
PLAY STOP suspends play event trapping

■ **Remarks**

If a **PLAY OFF** statement has been executed the **GOSUB** is not performed and is not remembered.

If a **PLAY STOP** statement has been executed the **GOSUB** is not performed, but will be performed as soon as a **PLAY ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **PLAY STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine will automatically perform a **PLAY ON** statement unless an explicit **PLAY OFF** was performed inside the subroutine.

The **RETURN** syntax

RETURN *linenumber*

may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB, WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

PLAY Statement

PLAY *"subcommand-string"*

■ **Action**

Plays music as specified by *subcommand-string*

■ **Remarks**

The *subcommand-string* is one or more of the subcommands listed below.

PLAY uses a concept similar to that in **DRAW** in that it embeds a music macro language string (described below) into one statement. A set of subcommands, used as part of the **PLAY** statement, specifies a particular action.

■ **Change Octave**

Increments octave. Octave will not advance beyond 6.

Decrements octave. Octave will not drop below 0.

■ **Tone**

O *n*   Sets the current octave. There are seven octaves, numbered 0 through 6.

A-G   Plays a note in the range A-G. A "#" or a "+" after the note specifies sharp; a "-" specifies flat.

N *n*   Plays note *n*. The range for *n* is 0 through 84 (in the 7 possible octaves, there are 84 notes); *n* = 0 means a rest.

■ **Duration**

L *n*   Sets the length of each note. L 4 is a quarter note, L1 is a whole note, etc. The *n* argument must be in the range 1 through 64.

The length may also follow the note when a change of length only is desired for a particular note. In this case, A 16 is equivalent to L 16 A.

MN   Sets "music normal" so that each note will play 7/8 of the time determined by the length (L).

ML   Sets "music legato" so that each note will play the full period set by length (L).

MS   Sets "music staccato" so that each note will play 3/4 of the time determined by the length (L).

■ Tempo

P n   Specifies a pause, ranging from 1 through 64. This option corresponds to the length of each note, set with L n.

T n   Sets the "tempo," or the number of L 4's in one minute. The n may range from 32 through 255. The default is 120.

■ Operation

MF   Sets music PLAY and SOUND statements to run in the foreground. That is, each subsequent note or sound will not start until the previous note or sound has finished. This is the default setting.

MB   Sets music PLAY and SOUND statements to run in the background. That is, each note or sound is placed in a buffer allowing the BASIC program to continue executing while the note or sound plays in the background.

■ Substring

X   *string*

Executes a substring. Because of the slow clock interrupt rate, some notes will not play at higher tempos (L 64 at T 255, for example).

PLAY Statement

■ Suffixes

or +   Follows a specified note, and turns it into a sharp.

Follows a specified note, and turns it into a flat.

A period after a note causes the note to play 3/2 times the length determined by L (length) times T (tempo). Multiple periods can appear after a note. Each period adds a length equal to one half the length of the previous period. For example, A. equals 1 + 1/2, or 3/2; A.. is 1 + 1/2 + 1/4, or 7/4; A... is 1 + 1/2 + 1/4 + 1/8, or 15/8; and so on. Periods can appear after a pause (P). In this case, the pause length can be scaled in the same way notes are scaled.

■ Example 1

This example will play the beginning of the first movement of Beethoven's Fifth Symphony:

```
100 LISTEN$ = "T180 O2 P2 P8 L8 GGG L2 E-"
110 FATE$ = "P24 P8 L8 FFF L2 D"
120 PLAY LISTEN$ + FATE$
```

■ Compiler/Interpreter Differences

In programs compiled with the GW-BASIC Interpreter, you should use the

VARPTR$(*variable*)

form for variables. For example, you should change interpreter statements such as

```
PLAY "XA$"
PLAY "O=I"
```

to

```
PLAY "X" + VARPTR$ (A$)
PLAY "O=" + VARPTR$ (I)
```

for the compiler.

■ Example 2

The following example uses ">" to play the scales from octave 0 to octave
6, then reverses with "<" to play the scales from octave 6 to octave 0:

```
SCALES$ = "CDEFGAB"
PLAY "OO X" + VARPTR$ (SCALE$)
FOR I = 1 TO 6
    PLAY ">X" + VARPTR$ (SCALE$)
NEXT
PLAY "O6 X" + VARPTR$ (SCALE$)
FOR I = 1 TO 6
    PLAY "<X" + VARPTR$ (SCALE$)
NEXT
```

PMAP Function

■ Syntax

PMAP *expression, function*

■ Action

Maps world coordinate expressions to physical locations or maps physical expressions to a world coordinate location

function =

0	Maps world expression to physical x coordinate.
1	Maps world expression to physical y coordinate.
2	Maps physical expression to world x coordinate.
3	Maps physical expression to world y coordinate.

■ Remarks

The four **PMAP** functions allow the user to find equivalent point locations between the world coordinates created with the **WINDOW** statement and the physical coordinate system of the screen or viewport as defined by the **VIEW** statement.

■ Examples

If a user had defined

```
WINDOW SCREEN (80,100) - (200,200)
```

then the upper left coordinate of the window would be (80,100) and the lower right would be (200,200). The screen coordinates may be (0,0) in the upper left hand corner and (639,199) in the lower right. Then

```
 X = PMAP (80,0)
```

will return the screen x coordinate of the window x coordinate 80, which is 0.

The **PMAP** function in the statement:

```
Y = PMAP(200,1)
```

will return the screen y coordinate of the window y coordinate 200, which is 199.

The **PMAP** function in the statement:

```
X = PMAP(619,2)
```

will return the world x coordinate that corresponds to the screen or viewport x coordinate 619, which is 199.

The **PMAP** function in the statement:

```
Y = PMAP(100,3)
```

will return the world y coordinate that corresponds to the screen or viewport y coordinate 100, which is 140.

POINT Function

■ Syntax

POINT (*xcoordinate,ycoordinate*)
POINT (*function*)

■ Action

POINT allows the user to read the color number of a pixel from the screen.

■ Remarks

The *xcoordinate* and *ycoordinate* are the coordinates of the pixel that is to be referenced.

If the specified point is out of range, the value -1 is returned.

POINT with one argument allows the user to retrieve the current graphics cursor coordinates. Therefore:
function =

POINT(0)	Returns the current physical *x* coordinate.
POINT(1)	Returns the current physical *y* coordinate.
POINT(2)	Returns the current logical *x* coordinate. If the **WINDOW** statement has not been used, this will return the same value as the **POINT(0)** function.
POINT(3)	Returns the current logical *y* coordinate if **WINDOW** is active, else returns the current physical *y* coordinate as in 1 above.

where the physical coordinate is the coordinate on the screen or current viewport.

■ Examples

```
10 SCREEN 1
20 C=3
30 PSET (10,10),C
40 IF POINT(10,10)=C THEN PRINT "This point is color "·C

5  SCREEN 2
```

```
10 IF POINT(i,i)<>0 THEN PRESET (i,i)
15 ELSE PSET (i,i)          'invert current state of a point

20 PSET (i,i),1-POINT(i,i) 'another way to invert a point
30 REM                     'if the system is black and white.
```

POKE Statement

■ Syntax

POKE *address,byte*

■ Action

Writes a byte into a memory location

■ Remarks

The arguments *address* and *byte* are integer expressions. The expression *address* represents the address of the memory location and *byte* is the data byte. The *byte* must be in the range 0 to 255.

The *address* must be in the range -32768 to 65535. The *address* is the offset from the current segment, which was set by the last **DEF SEG** statement. For interpretation of negative values of *address* see "VARPTR Function."

The complementary function to **POKE** is **PEEK**.

Warning

Use **POKE** carefully. If it is used incorrectly, it can cause the system to crash.

■ See Also

DEF SEG, PEEK, VARPTR

■ Example

10 POKE &H5A00, &HFF

■ **Syntax**

POS(I)

■ **Action**

Returns the current horizontal (column) position of the cursor

■ **Remarks**

The leftmost position is 1. I is a dummy argument. To return the current vertical line position of the cursor, use the CSRLIN function.

■ **Example**

```
IF POS(X) > 60 THEN BEEP
```

■ **See Also**

CSRLIN, LPOS

PRESET Statement

■ **Syntax**

PRESET [STEP](*xcoordinate,ycoordinate*) [,*color*]

■ **Action**

Draws a specified point on the screen

■ **Remarks**

PRESET works exactly like **PSET** except that if the *color* is not specified, the background color is selected.

The *xcoordinate* and *ycoordinate* specify the pixel that is to be set.

The *color* is the color number that is to be used for the specified point.

The **STEP** option, when used, indicates the given x and y coordinates will be relative, not absolute. That means the x and y are distances from the most recent cursor location, not distances from the (0,0) screen coordinate. If a coordinate is outside the current viewport, no action is taken, nor is an error message given.

Coordinates can be shown as absolutes, as in the above syntax, or the **STEP** option can be used to reference a point relative to the most recent point used. For example, if the most recent point referenced were (10,10),

STEP (10,5)

would reference the point at (20,15).

■ **See Also**

PSET

■ **Example**

The following example draws a line from (0,0) to (100,100) and then erases that line by overwriting it with the background color:

```
5  SCREEN 2  'Draw a line from (0,0) to (100,100)
10 FOR I=0 TO 100
```

```
20    PRESET (I,I),1
30 NEXT
35 REM Now erase that line
40 FOR I=0 TO 100
50    PRESET STEP (-1,-1)
60 NEXT
```

PRINT Statement

■ Syntax

PRINT [[*expressionlist*]]

■ Action

Outputs data on the screen

■ Remarks

If *expressionlist* is omitted, a blank line is printed. If *expressionlist* is included, the values of the expressions are printed on the screen. The expressions in the list may be numeric and/or string expressions. (String literals must be enclosed in quotation marks.)

A question mark (?) can be used as a form of shorthand by the user. It will be interpreted as the word PRINT, and will appear as PRINT in subsequent listings.

■ Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next **PRINT** statement begins printing on the same line, spacing according to instructions. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is wider than the screen width, GW-BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8 is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format-no less accurately than they can be represented in the scaled format, are output

using the unscaled format. For example, 1D-15 is output as
.0000000000000001 and 1D-16 is output as 1D-16.

With the interpreter, a question mark may be used in place of the word
PRINT in a **PRINT** statement.

■ **Example 1**

In this example, the commas in the **PRINT** statement cause each value to
be printed at the beginning of the next print zone.

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X_5
30 END
```

Output:

```
 10             0            -25           3125
```

■ **Example 2**

In this example, the semicolon at the end of line 20 causes both PRINT
statements to be printed on the same line. Line 40 causes a blank line to be
printed before the next prompt.

```
5   PRINT "Input 0 to end."
10  INPUT X
15  IF X = 0 THEN END
20  PRINT X "SQUARED IS" X^2 "AND";
30  PRINT X "CUBED IS" X^3
40  PRINT
50  GOTO 5
```

Output:

```
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

? 0
```

PRINT Statement

■ **Example 3**

In this example, the semicolons in the **PRINT** statement cause each value
to be printed immediately after the preceding value. (Remember, a number
is always followed by a space, and positive numbers are preceded by a
space.) **PRINT.**

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 PRINT J;K;
50 NEXT X
```

Output:

```
 5   10   10   20   15   30   20   40   25   50
```

■ **Syntax**

PRINT# *filenumber,*[[USING *stringexpression;*]] *expressionlist*

■ **Action**

Writes data to a sequential file.

■ **Remarks**

The *filenumber* is the number used when the file was opened for output. The *stringexpression* consists of formatting characters as described in "PRINT USING Statement." The expressions in *expressionlist* are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data. An image of the data is written to the file, just as it would be displayed on the terminal screen with a PRINT-statement. For this reason, care should be taken to delimit the data, so that it will be input correctly.

In the list of expressions, numeric expressions should be delimited by semi-colons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the file, use explicit delimiters in the list. of expressions.

For example, assume the following is true:

```
A$="CAMERA"
B$="93604-1"
```

Then the statement

```
PRINT#1,A$;B$
```

will write

```
CAMERA93604-1
```

to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to the file now is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the file surrounded by explicit quotation marks with CHR$(34).

For example, assume that the following is true:

```
A$="CAMERA, AUTOMATIC"
B$="    93604-1"
```

Then the statement

```
PRINT#1,A$;B$
```

will write the following image to file:

```
CAMERA, AUTOMATIC    93604-1
```

And the statement

```
INPUT#1,A$,B$
```

will input

```
CAMERA
AUTOMATIC93604-1
```

to A$ and B$, respectively. To separate these strings properly in the file, write double quotation marks to the file image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$ ;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC""    93604-1"
```

And the statement

```
INPUT#1,A$,B$
```

will input

```
"CAMERA, AUTOMATIC"
"   93604-1"
```

to A$ and B$, respectively.

The **PRINT#** statement may also be used with the **USING** option to control the format of the file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

■ **See Also**

WRITE#

PRINT USING Statement

■ **Statement Syntax**

PRINT USING *stringexpression;expressionlist*

■ **Action**

Prints strings or numbers using a specified format.

■ **Remarks**

The *expressionlist* contains the string expressions or numeric expressions that are to be printed, separated by semicolons.

The *stringexpression* is a string literal (or variable) composed of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

■ **Formatting Characters: String Fields**

When **PRINT USING** is used to print strings, one of three formatting characters may be used to format the string field:

!　　　Specifies that only the first character in the given string is to be printed.

\ *n spaces*\
　　　Specifies that 2 + n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

&　　　Specifies a variable length string field. When the field is specified with the ampersand (&), the string is output without modification.

Here is an example of how the three string formatting characters (!,\ \,&) affect printed output:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\    \";A$;B$;"!!"
60 PRINT USING "!";A$;
70 PRINT USING "&";B$
```

Output:

```
LO
LOOKOUT
LOOK OUT  !!
LOUT
```

■ Formatting Characters: Numeric Fields

When **PRINT USING** is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

■ Example 1

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20    5.30   66.79    0.23
```

In the last example above, three spaces were inserted at the end of the format string to separate the printed values on the line.

PRINT USING Statement

■ **Example 2**

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95    +2.40    +55.60    -0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
68.95-   22.45    7.01-
```

+	A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.
−	A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

■ **Example 3**

```
PRINT USING "**#.#   ";12.39,-0.9,765.1
*12.4   *-0.9   765.1

PRINT USING "$$###.##";456.78
 $456.78

PRINT USING "**$##.##";2.34
***$2.34
```

**	A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The double asterisk also specifies positions for two more digits.
$$	A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign.
**$	The double asterisk dollar sign combines the effects of the double asterisk and double dollar sign symbols. Leading spaces are asterisk-filled and a dollar sign is printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.
	The exponential format cannot be used with **$. When negative numbers are printed, the minus sign will appear immediately to the left of the dollar sign.

■ Example 4

```
PRINT USING "####.##,";1234.5
1234.50,

PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";-888888
.8889E+06-

PRINT USING "+.##^^^^";123
+.12E+03

PRINT USING "_!##.##_!";12.34
!12.34!

PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

,   A comma to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with exponential (^^^^)format.

^ ^ ^ ^   Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

_   An underscore in the format string causes the next character to be output as a literal character.

The literal character itself may be an underscore by placing __ in the format string.

%   If the number to be printed is larger than the specified
           numeric field, a percent sign is printed in front of the
           number. If rounding causes the number to exceed the field,
           a percent sign will be printed in front of the rounded
           number.

           If the number of digits specified exceeds 24, an "Illegal
           function call" error results.

■ **Syntax**

PSET [[**STEP**]]($xcoordinate, ycoordinate$) [[, $color$]]

■ **Remarks**

The *xcoordinate* and *ycoordinate* specify the point on the screen to be colored.

The *color* is the number of the color to be used.

The **STEP** option, when used, indicates the given x and y coordinates will be relative, not absolute. That means the x and y are distances from the most recent cursor location, not distances from the (0,0) screen coordinate. When GW-BASIC scans coordinate values, it will allow them to be beyond the edge of the screen (the size of the screen is dependent on display hardware being used, and can be adjusted with the **WIDTH** statement). However, values outside the integer range -32768 to 32767 will cause an "Overflow" error.

Coordinates can be shown as offsets by using the **STEP** option to reference a point relative to the most recent point used. The syntax of the **STEP** option is:

STEP ($xoffset, yoffset$)

For example, if the most recent point referenced were (0,0),

PSET STEP (10,0)

would reference a point at offset 10 from x and offset 0 from y.

The coordinate (0,0) is always the upper left corner of the screen.

PSET allows the *color* to be left off the command line. If it is omitted, the default is the foreground color.

■ **Example**

This example draws a line from (0,0) to (100,100) and then erases that line by writing over it with the background color:

5   SCREEN 2   'Draw a line from (0,0) to (100,100).

PSET Statement

```
10 FOR I=0 TO 100
20     PSET (I,I)
30 NEXT I
35 REM Now erase that line
40 FOR I=0 TO 100
50     PSET STEP (-1,-1),0
60 NEXT I
```

■ **Syntax**

PUT (x1,y1),*arrayname*[[,**actionverb**]]

■ **Action**

The **GET** and **PUT** statements are used together to transfer graphic images to and from the screen. The syntax for **PUT** is as follows:

GET (x1,y1)-(x2,y2),*arrayname*

The **GET** statement transfers the screen image bounded by the rectangle described by the specified points into the array.

The **PUT** statement transfers the image stored in the array onto the screen.

The *actionverb* specifies the interaction between the stored image and the one already on the screen.

■ **Remarks**

The coordinates *x1,y1* in the **PUT** statement specify the point where a stored image is to be displayed on the screen. The specified point is the coordinate of the top left corner of the image. If the image to be transferred is too large to fit in the current viewport, an "Illegal function call" error will result.

The *actionverb* is one of: **PSET, PRESET, AND, OR,** or **XOR** as described below:

PSET	Transfers the data point by point onto the screen. Each point has the exact color attribute it had when it was taken from the screen with a **GET.**
PRESET	The same as **PSET** except that a negative image (black on white) is produced.
AND	Used when the image is to be transferred over an existing image on the screen. The resulting image is the product of the logical AND expression; points that had the same color in both the existing image and the **PUT** image will remain the same color, points that do not have the same color in both the existing image and the **PUT** image, will not.

> **OR**   Used to superimpose the image onto an existing image.
>
> **XOR**   A special mode often used for animation. **XOR** causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like that of the cursor. When an image is **PUT** against a complex background twice, the background is restored unchanged. This allows a user to move an object around the screen without erasing the background.

The default *actionverb* is **XOR**.

One of the most useful things that can be done with **GET** and **PUT** is animation. Animation for an object is performed as follows:

1. **PUT** the object on the screen.

2. Recalculate the new position of the object.

3. **PUT** the object on the screen a second time at the old location (using **XOR**) to remove the old image.

4. Go to step 1, but this time **PUT** the object at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the **PSET** action verb. The idea is to leave a border around the image when it is first gotten that is as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points left by the previous **PUT**. This method may be somewhat faster than the method using **XOR** described above, since only one **PUT** is required to move an object (although you must **PUT** a larger image).

It is possible to examine the x and y dimensions and even the data itself if an integer array is used. With the interpreter, the x dimension is in element 0 of the array, and the y dimension is found in element 1. (However, this will not always be true for the compiler.) Remember that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

■ **Syntax**

PUT [[#]]*filenumber*[[,*recordnumber*]]

■ **Action**

Writes a record from a random buffer to a random access file

■ **Remarks**

The *filenumber* is the number under which the file was opened. If *recordnumber* is omitted, the record will assume the next available record number (after the last PUT or GET). The largest possible record number is 16,777,215. The smallest record number is 1.

The GET and PUT statements allow fixed-length input and output for GW-BASIC .COM files. However, because of the low performance associated with telephone line communications, it is recommended that you do not use GET and PUT for telephone communication.

Note

    LSET, RSET, PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE#, Microsoft GW-BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

■ **Example**

100 PUT 1, A$, B$, C$

RANDOMIZE Statement

■ **Syntax**

RANDOMIZE[[*expression*]]

■ **Action**

Reseeds the random number generator

■ **Remarks**

If *expression* is omitted, GW-BASIC suspends program execution and asks
for a value by printing

```
Random Number Seed (-32768 to 32767)?
```

before executing **RANDOMIZE**.

If expression is a variable, the value of that variable is used to seed the ran-
dom numbers.

If *expression* is the word TIMER then the **TIMER** function is used to pass a
random number seed.

If the random number generator is not reseeded, the **RND** function returns
the same sequence of random numbers each time the program is run. To
change the sequence of random numbers every time the program is run,
place a **RANDOMIZE** statement at the beginning of the program and
change the argument with each run.

■ **Example**

In this example, identical values input after the randomize prompt result in
identical sequences of random numbers:

```
5    FOR J = 1 TO 3
10     RANDOMIZE
20     FOR I=1 TO 5
30        PRINT RND;
40     NEXT I
45     PRINT
50 NEXT J
```

Output:

```
Random Number Seed (-32768 to 32767)? 3

.885982   .4845668   .586328   .1194246   .7039225

Random Number Seed (-32768 to 32767)? 4

.803506   .1625462   .929364   .2924443   .322921

Random Number Seed (-32768 to 32767)? 3

.885982   .4845668   .586328   .1194246   .7039225
```

Note that the numbers your program produces may not be the same as the ones shown here.

READ Statement

■ Syntax

READ *variablelist*

■ Action

Reads values from a **DATA** statement and assign them to variables

■ Remarks

A **READ** statement must always be used in conjunction with a **DATA** statement. **READ** statements assign variables to **DATA** statement values on a one-to-one basis. **READ** statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a syntax error will result.

A single **READ** statement may access one or more **DATA** statements (they will be accessed in order), or several **READ** statements may access the same **DATA** statement. If the number of variables in *variablelist* exceeds the number of elements in the **DATA** statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the **DATA** statement(s), subsequent **READ** statements will begin reading data at the first unread element. If there are no subsequent **READ** statements, the extra data is ignored.

To reread **DATA** statements from the start, use the **RESTORE** statement.

■ See Also

DATA, RESTORE

■ Example 1

This program segment reads the values from the **DATA** statements into the array A. After execution, the value of A(1) will be 3.08, the value of A(2) will be 5.19, and so on.

.
.
.
```
80  FOR I=1 TO 10
```

```
90      READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
```
.
.
.

■ **Example 2**

This program reads string and numeric data from the **DATA** statement in line 30:

```
10 PRINT "CITY", "STATE"  " ZIP"
20 READ C$,S$,Z$
30 DATA "DENVER,",  "COLORADO","80211"
40 PRINT C$,S$,Z$
```

Output:

```
CITY            STATE          ZIP
DENVER          COLORADO       80211
```

REM Statement

■ **Syntax**

REM *remark*

■ **Action**

Allows explanatory remarks to be inserted in a program

■ **Remarks**

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a **GOTO** or **GOSUB** statement. Execution will continue with the first executable statement after the REM statement.

A remark may be added to the end of a line by preceding the remark with a single quotation mark (') instead of REM.

Important

Do not use the single quotation form of the REM statement in a **DATA** statement because it will be considered legal data.

■ **Example**

The following fragments demonstrate the two different ways to add remarks to a program:

```
.
.
.
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
.
.
.
```

or

.
.
.
```
100 DIM V(20)      'Calculate average velocity
120 FOR I=1 TO 20
130    SUM=SUM+V(I)
140 NEXT I
```
.
.
.

RENUM Command

■ **Syntax**

RENUM [[[*newnumber*]][,[*oldnumber*]][,*increment*]]]

■ **Action**

Renumbers program lines

■ **Remarks**

The *newnumber* is the first line number to be used in the new sequence. The default is 10. The *oldnumber* is the line in the current program where renumbering is to begin. The default is the first line of the program. The *increment* is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following **GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB,** and **ERL** statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number in *xxxxx*" is printed. The incorrect line number reference is not changed by **RENUM**, but line number *yyyyy* may be changed.

Note

RENUM cannot be used to change the order of program lines (for example,

RENUM 15,30

when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result in such a case.

■ **Examples**

Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10:

RENUM

Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50:

RENUM 300,,50

Renumbers the lines from 900 up so they start with line number 1000 and are numbered in increments of 20:

RENUM 1000,900,20

■ **Compiler Differences**

The **RENUM** command is not supported by the compiler.

RESET Command

■ **Syntax**

RESET

■ **Action**

Closes all files

■ **Remarks**

RESET closes all open files and forces all blocks in memory to be written to disk. Thus, if the machine loses power, all files will be properly updated.

All files must be closed before a disk is removed from its drive.

■ **Example**

```
998 RESET
999 END
```

■ **Syntax**

RESTORE [*linenumber*]

■ **Action**

Allows **DATA** statements to be reread from a specified line.

■ **Remarks**

After a **RESTORE** statement without a specified *linenumber* is executed, the next **READ** statement accesses the first item in the first **DATA** statement in the program.

If *linenumber* is specified, the next **READ** statement accesses the first item in the specified **DATA** statement.

■ **Example**

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
.
.
.
```

RESUME Statement

■ Syntax

RESUME *linenumber*
RESUME [0]
RESUME NEXT

■ Action

Continues program execution after an error recovery procedure has been performed

■ Remarks

Any one of the three syntaxes shown above may be used, depending upon where execution is to resume:

RESUME [0]	Execution resumes at the statement that caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one that caused the error.
RESUME *linenumber*	Execution resumes at *linenumber*.

A **RESUME** statement that is not in an error handling routine causes a "RESUME without error" message to be printed.

■ Compiler/Interpreter Differences

The compiler supports the following extended syntax:

RESUME { *linenumber*,*linelabel*}

In addition, this is a statement that may require modification of interpreted BASIC programs when used with the compiler. In the compiler, if an error occurs in a single-line function, both **RESUME** and **RESUME NEXT** attempt to resume program execution at the line containing the function.

Note

Statements containing error-handling routines should be compiled with either the /X switch or the /E option.

Considerable extra code is required to support both **RESUME** and **RESUME NEXT**. If you can rewrite your programs so that they instead contain **RESUME** *linenumber* or **RESUME** *linelabel* statements, you can compile with the /E option, thus making user code significantly smaller.

■ **Example 1**

This example has an error-handling routine that starts at line 500. If "fntest" tries to evaluate the square root of a negative number, line 500 prints its error message. In the interpreter, control then resumes at line 150, and the **FOR...NEXT** loop continues. In the compiler, however, control resumes at line 110, which contains the function definition. This causes execution to go into an endless loop. In programs such as this, you should change **RESUME NEXT** statements to **RESUME** *linenumber* or **RESUME** *linelabel*.

```
100  on error goto 500
110  def fntest(a) = 1 - sqr(a)
120  for i = 4 to -2 step -1
130      print i, fntest(i)
140  next
150  end
500  print "No negative arguments"
510  resume next
```

RESUME Statement

```
Interpreter output:              Compiler output:
 4       -1                        4       -1
 3       -.7320509                 3       -.7320509
 2       -.4142136                 2       -.4142136
 1        O                        1        O
 O        1                        O        1
-1       No negative arguments    -1       No negative arguments
-2       No negative arguments     4       -1
                                   3       -.7320509
                                   2       -.4142136
                                   .
                                   .
                                   .
```

■ Example 2

This example has an error-handling routine that starts at line "errorhandle". If the error is the one that the user has defined as 200 (file is empty), then the routine asks for a new file name for input, and resumes execution at line "continue". If the error is something else, BASIC prints the message "Unprintable error".

```
let condition% = 0
on error goto errorhandle
open "phone" for input as #1
if lof(1) = 0 then error 200
continue:
.
.
.
end
errorhandle: rem ** Error handling fragment **
    rem ** Next line prints "Unprintable error" if true **
    if err <> 200 then error err _
    else _
    rem ** Routine for error 200 **
        print "Phone number file is empty"
        input "Name of file that has information"; fil$
        close #1
        open fil$ for input as #1
        resume continue
```

■   **Statement Syntax**

RETURN [[*linenumber*]]

■   **Action**

Returns execution control from a subroutine

■   **Remarks**

If no *linenumber* is given, execution begins with the statement immediately following the last executed **GOSUB** statement. Otherwise, the *linenumber* in the **RETURN** statement causes an unconditional return from a **GOSUB** subroutine to the specified line.

■   **See Also**

GOSUB

■   **Compiler/Interpreter Differences**

Since the compiler allows the use of line labels, the extended syntax for **RETURN** in the compiler is as follows:

RETURN [[{ *linenumber*, *linelabel*}]]

RIGHT$ Function

■ **Syntax**

RIGHT$($x$$,n)

■ **Action**

Returns the rightmost n characters of string $x$$.

■ **Remarks**

If n is equal to the number of characters in $x$$ (that is, n=LEN($X$$)), then RIGHT$ returns $x$$. If n=O, the null string (length zero) is returned.

■ **Example**

```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,5)
```

Output:

```
BASIC
```

■ **See Also**

LEFT$, MID$

■ **Syntax**

RMDIR *pathname*

■ **Action**

Removes an existing directory

■ **Remarks**

The *pathname* is the name of the directory which is to be deleted. **RMDIR** works exactly like the operating system command of the same name. The *pathname* must be a string of less than 128 characters.

The *pathname* to be removed must be empty of any files except the working directory ('.') and the parent directory ('..') or else a "Path not found" or a "Path/File Access error" is given.

■ **Example**

```
RMDIR "SALES"
```

In this statement, the SALES directory on the current drive is to be removed.

RSET Statement

■ **Syntax**

RSET *stringvariable*=*x*$
LSET *stringvariable*=*x*$

■ **Action**

Moves data from memory to a random file buffer (in preparation for a **PUT** statement) or left- or right-justifies the value of a string in a string variable

■ **Remarks**

If *x*$ requires fewer bytes than were fielded to *stringvariable*, **LSET** left-justifies the string in the field, and **RSET** right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are **LSET** or **RSET**.

■ **Example**

See example under the "LSET Statement."

■ **Syntax**

RND[(n)]

■ **Action**

Returns a random number between 0 and 1

■ **Remarks**

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see the "RANDOM-IZE Statement" for more information). However, $n < 0$ always restarts the same sequence for any given n.

If $n > 0$ or if n is omitted then **RND** generates the next random number in the sequence. If $n = 0$, then **RND** repeats the last number generated.

■ **Example**

```
10 FOR I=1 TO 5
20   PRINT INT(RND*100);
30 NEXT I
```

Output:

```
 24  30  31  51   5
```

RUN Command

■ **Syntax**

RUN [[*linenumber*]]
RUN "*filespec*"[[,R]]

■ **Action**

Executes the program currently in memory, or loads a file into memory and runs it

■ **Remarks**

For a program currently in memory, if *linenumber* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after a RUN statement is executed.

For running a program not in memory, the *filespec* is an optional device specification followed by a file name or path name that conforms to operating system naming conventions for file names. BASIC appends the default filename extension .BAS if the user specifies no extensions, and the file has been saved to disk.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the ,R option, all data files remain open.

■ **Example**

RUN "NEWFIL",R

■ **Compiler/Interpreter Differences**

The following syntaxes are supported in the compiler:

RUN [[*linenumber*]]
RUN "*filespec*"

The first syntax *restarts* the program currently in memory. If *linenumber* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number, or the first executable line, if you use line labels.

With the second form of the syntax, the named file is loaded from a device into memory and run. If there is a program in memory when the command executes, the original program is no longer in memory.

In the second syntax, the *filespec* must be that used when the file was saved. This string expression can contain a path.

The default filename extension in the compiler is ".EXE", just as the default extension in the interpreter is ".BAS". Therefore, if "CATCHALL" is the name of both an executable file and a BASIC source file, the command

```
RUN "CATCHALL"
```

will work correctly in both the interpreter and the compiler.

RUN closes all open files and clears the current contents of memory before loading the designated program. The compiler does not support the interpreted BASIC ,R option, which allows all open data files to remain open. If you want to run a new file, yet leave all data files open, use **CHAIN**.

Note

> **RUN** is used to invoke .EXE files created by the compiler, or .EXE and .COM files created by other languages; unlike interpreted programs, compiled programs cannot directly execute ".BAS" source files.

■   **See Also**

CHAIN

■   **Example 2**

The following line in a program transfers program control to the file "PRICE", after closing all open files and deleting the current memory contents:

```
RUN "PRICE"
```

■ Example 3

The following command runs the program currently in memory, starting with line 950:

RUN 950

■ **Syntax**

SAVE *filespec*[[{,A|,P}]]

■ **Action**

Saves a program file

■ **Remarks**

For saving a program in memory, the *filespec* is an optional device specification followed by a file name or path name that conforms to operating system naming conventions for file names. BASIC appends the default file name extension .BAS if the user specifies no extensions, and the file has been saved to disk.

The ,A option saves the file in ASCII format. If the ,A option is not specified, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some actions require that files be in ASCII format. For instance, the **MERGE** command requires an ASCII format file.

The **P** option protects the file by saving it in an encoded binary format. When a protected file is later loaded any attempt to list or edit it will fail.

■ **Examples**

Saves the program COM2 in ASCII format:

SAVE "COM2",A

Saves the program PROG.BAS as a protected file which cannot be altered:

SAVE "PROG",P

■ **Compiler Differences**

The compiler does not support the **SAVE** command.

SCREEN Function

■ Syntax

SCREEN(*row,column*[[,*z*]])

■ Action

Reads a character or its color from a specified screen location

■ Remarks

The *row* is a valid numeric expression returning an unsigned integer.

The *column* is a valid numeric expression returning an unsigned integer.

The argument *z* is a valid numeric expression.

The ordinate of the character at the specified coordinates is stored in the numeric variable. If the optional parameter z is given and is non-zero, the color attribute for the character is returned instead.

■ Examples

If the character at (10,10) is A, then the function would return 65, the ASCII code for A:

```
100  X=SCREEN(10,10)
```

Returns the color attribute of the character in the upper left corner of the screen:

```
100  X=SCREEN(1,1,1)
```

■ **Syntax**

SCREEN [[*mode*] [, [[*colorswitch*]] [, [[*apage*]] [, [[*vpage*]]]

■ **Action**

Sets the specifications for the display screen

■ **Remarks**

The **SCREEN** statement is chiefly used to select a screen mode appropriate to a particular display hardware configuration. Supported hardware configurations and screen modes are described below.

MDPA with Monochrome Display: Mode 0

The IBM Monochrome Display and Printer Adapter (MDPA) is used to connect only to a Monochrome Display. Programs written for this configuration must be text mode only.

CGA with Color Display: Modes 0, 1, and 2

The IBM Color Graphics Adapter (CGA) and Color Display are typically paired with each other. This hardware configuration permits running of text mode programs, and both medium resolution and high resolution graphics programs.

EGA with Color Display: Modes 0, 1, 2, 7, and 8

The five screen modes 0, 1, 2, 7, and 8 allow you to interface to the IBM Color Display when it is attached to an IBM EGA (Enhanced Graphics Adapter). If EGA switches are set for CGA compatability, programs written for modes 1 and 2 will run just as if they would with the CGA. Modes 7 and 8 are similar to modes 1 and 2, except that a wider range of colors are available in modes 7 and 8.

SCREEN Statement

EGA with Enhanced Display: Modes 0, 1, 2, 7, and 8

With the EGA/Enhanced Display configuration, modes 0, 1, 2, 7, and 8 are identical to their EGA/Color Display counterparts. Possible differences:

1. In mode 0, the border color may not be the same because the border cannot be set on an Enhanced Color Display when it is in 640 x 350 text mode.

2. The quality of the text is better on the Enhanced color display (8 x 14 character box on Enhanced color display versus 8 x 8 character box on color display).

EGA with Enhanced Display: Mode 9

The full capability of the IBM Enhanced Display is taken advantage of in this mode. The highest resolution possible in any of the modes is with this hardware configuration. Programs written for this mode will not work for any other hardware configuration.

EGA with Monochrome Display: Mode 10

The Monochrome Display can be used to display monochrome graphics at a very high resolution in this mode. Programs written for this mode will not work for any other hardware configuration.

■ Arguments

The *mode* argument is an integer expression with legal values 0, 1, 2, 7, 8, 9, 10. All other values are illegal. Your selection of a mode argument depends primarily on your program's anticipated display hardware, as described above.

Each of the **SCREEN** modes is described individually in the following paragraphs.

SCREEN 0

- Text mode only
- Either 40 x 25 or 80 x 25 text format with character box size of 8 x 8 (8 x 14 with EGA)

- Assignment of 16 colors to any of 2 attributes
- Assignment of 16 colors to any of 16 attributes (with EGA)

SCREEN 1

- 320 x 200 pixel medium resolution graphics
- 80 x 25 text format with character box size of 8 x 8
- Assignment of 16 colors to any of 4 attributes
- Supports both EGA and CGA

SCREEN 2

- 640 x 200 pixel medium resolution graphics
- 40 x 25 text format with character box size of 8 x 8
- Assignment of 16 colors to any of 2 attributes
- Supports both EGA and CGA

SCREEN 7

- 320 x 200 pixel medium resolution graphics
- 40 x 25 text format with character box size of 8 x 8
- 2, 4, or 8 memory pages with 64K, 128K, or 256K of memory, respectively, installed on the EGA
- Assignment of any of 16 colors to 16 attributes
- EGA required .

SCREEN 8

- 640 x 200 pixel high resolution graphics
- 80 x 25 text format with character box size of 8 x 8
- 1, 2, or 4 memory pages with 64K, 128K, or 256K of memory, respectively, installed on the EGA

SCREEN Statement

- Assignment of any of 16 colors to 16 attributes
- EGA required

SCREEN 9

- 640 x 350 pixel enhanced resolution graphics
- 80x25 text format with character box size of 8 x 14,
- Assignment of any of 64 colors to either 4 or 16 attributes, (With 64K of EGA memory, there are 4 attributes; with greater than 64K, 16 attributes)
- Two display pages if 256K of EGA memory installed
- EGA required

SCREEN 10

- 640 x 350 enhanced resolution graphics
- 80 x 25 text format with character box size of 8 x 14
- Two display pages if 256K of EGA memory installed
- Assignment of up to 9 pseudo-colors to 4 attributes
- EGA required

Table 6.2

Default Attributes: SCREEN 10, Monochrome Display

Attribute Value	Displayed Pseudo-Color
0	Off
1	On, normal intensity
2	Blink
3	On, high intensity

Table 6.3

Color Values: SCREEN 10, Monochrome Display

Color Value	Displayed Pseudo-Color
0	Off
1	Blink, Off to On
2	Blink, Off to High intensity
3	Blink, On to Off
4	On
5	Blink, On to High intensity
6	Blink, High intensity to Off
7	Blink, High intensity to On
8	High intensity

For composite monitors and TVs, the *colorswitch* is a numeric expression that is either true (non-zero) or false (zero). A value of zero disables color and permits display of black and white images only. A non-zero value permits color. The meaning of the *colorswitch* argument is inverted in SCREEN mode 0.

For hardware configurations that include an EGA and enough memory to support multiple screen pages, two arguments are available. These *apage* and *vpage* arguments determine the "active" and "visual" memory pages. The active page is the area in memory where graphics statements are written; the visual page is the area of memory that is displayed on the screen. Animation can be achieved by alternating display of graphics pages. The goal here is to display already completed graphics output on the visual page, while executing graphics statements in one or more active pages. A page is displayed only when graphics output to that page is complete. Thus the following is typical:

```
SCREEN 7,,1,2    'work ·in page 1, show page 2
.
.   Graphics output to page 1
.   while viewing page 2
.
SCREEN 7,,2,1    'work in page 2, show page 1
.
.   Graphics output to page 2
.   while viewing page 1
.
```

The number of pages available depends on the SCREEN mode and the amount of available memory, as described in the following table.

Table 6.4
SCREEN Mode Specifications

Mode	Resolution	Attribute Range	Color Range	EGA Memory	Pages	Page Size
0	40-column text	NA	0-15*	NA	1	2K
	80-column text	NA	0-15*	NA	1	4K
1	320 x 200	0-3$^\perp$	0-3	NA	1	16K
2	640 x 200	0-1$^\perp$	0-1	NA	1	16K
7	320 x 200	0-15	0-15	64K	2	32K
				128K	4	
				256K	8	
8	640 x 200	0-15	0-15	64K	1	64K
				128K	2	
				256K	4	
9	640 x 350	0-3	0-15	64K	1	64K
		0-15	0-63	128K	1	128K
		0-15	0-63	256K	2	
10	640 x 350	0-3	0-8	128K	1	128K
				256K	2	

$\perp$ Attributes applicable only with EGA
* Numbers in the range 16-31 are blinking versions of the colors 0-15.

▩ Attributes and Colors

For various screen modes and display hardware configurations, different attribute and color settings exist. (See the "PALETTE Statement" for a discussion of attribute and color number.) The majority of these attribute and color configurations are summarized in the following table.

Table 6.5

Default Attributes and Colors
For Most Screen Modes

Attribute Value for Mode			Color Display		Monochrome Display	
1,9	2	0,7,8,9+	#	Color	#	Pseudo-color
0	0	0	0	Black	0	Off
		1	1	Blue		(Underlined) (*)
		2	2	Green	1	On (*)
		3	3	Cyan	1	On (*)
		4	4	Red	1	On (*)
		5	5	Magenta	1	On (*)
		6	6	Brown	1	On (*)
		7	7	White	1	On (*)
		8	8	Gray	0	Off
		9	9	Light Blue		High intensity (underlined)
		10	10	Light Green	2	High intensity
1		11	11	Light Cyan	2	High intensity
		12	12	Light Red	2	High intensity
2		13	13	Light Magenta	2	High intensity
		14	14	Yellow	2	High intensity
3	1	15	15	High-intensity White	0	Off

* Off when used for background.
+ With EGA memory > 64K
# Only for mode 0 monochrome

SGN Function

SGN(expression)

■ **Action**

Indicates the value of the *expression*, relative to zero

■ **Remarks**

If *expression* > 0, **SGN**(*expression*) returns 1.

If *expression* = 0, then **SGN**(*expression*) returns 0.

If *expression* < 0, then **SGN**(*expression*) returns -1.

■ **Example**

This statement causes control to branch to 100 if X is negative, 200 if X is 0, and 300 if X is positive:

```
ON SGN(X)+2 GOTO 100,200,300
```

■ **Syntax**

SHELL [[*commandstring*]]

■ **Action**

Exits the BASIC program, runs a .COM, .EXE, or .BAT program (or a built-in DOS function such as DIR oor TYPE), and returns to the BASIC program at the line after the SHELL statement.

■ **Remarks**

A .COM, .EXE, or .BAT program or DOS function which runs under the SHELL statement is called a "child process." Child processes are executed by SHELL loading and running a copy of COMMAND.COM with the /C option. By using COMMAND in this way, command line parameters are passed to the child. Standard input and output may be redirected, and built in commands such as DIR, PATH, and SORT may be executed.

The *commandstring* must be a valid string expression containing the name of a program to run and (optionally) command arguments.

The program name in *commandstring* may have any extension you wish. If no extension is supplied, COMMAND.COM will look for a .COMfile, then a .EXEfile, and finally, a .BATfile. If COMMAND.COM is not found, SHELL will issue a "File not found" message. No error is generated if COMMAND.COM cannot find the file specified in *commandstring*.

Any text separated from the program name by at least one blank will be processed by COMMAND.COM as program parameters.

BASIC remains in memory while the child process is running. When the child finishes, BASIC continues.

SHELL with no *commandstring* will give you a new COMMAND.COM shell. You may now do anything that COMMAND.COM allows. When ready to return to BASIC, enter the DOS command, EXIT

SHELL Statement

■ **Examples**

```
SHELL   'get a new COMMAND
```

A>DATE

A>EXIT

Ok

Write some data to be sorted, **SHELL** sort to sort it, then read the sorted data to write a report:

```
900   OPEN "SORTIN.DAT" FOR OUTPUT AS 1
950   REM ** write data to be sorted
1000  CLOSE 1
1010  SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
1020  OPEN "SORTOUT.DAT" FOR INPUT AS 1
1030  REM ** Process the sorted data
```

SHELL to MS-DOS, sort the directory of the files on disk, store this data in FILES.DSK, then open this file:

```
10 SHELL "DIR | SORT >FILES.DSK"
20 OPEN "FILES.DSK" FOR INPUT AS 1
```

■ **Syntax**

SIN(*x*)

■ **Action**

Returns the sine of *x*, where *x* is in radians

■ **Remarks**

COS(X) = SIN(X+3.14159/2).

■ **Example**

```
PRINT SIN(1.5)
```

Output:

```
.9974951
```

■ **See Also**

COS

SOUND Statement

■ **Syntax**

SOUND *freq,duration* [,[*volume*][,[*voice*]]]

■ **Action**

Generates sound through the speaker

■ **Remarks**

The *freq* is the desired frequency in hertz (cycles/second). This must be a numeric expression returning an unsigned integer in the range 37 to 32,767.

The *duration* is the duration in clock ticks. Clock ticks occur 18.2 times per second. This must be a numeric expression returning an unsigned integer in the range 0 to 65535.

If the duration is zero, any current **SOUND** statement that is running is turned off. If no **SOUND** statement is currently running, a **SOUND** statement with a duration of zero has no effect.

■ **Compiler/Interpreter Differences**

The compiler supports the optional arguments *volume* and *voice*. The argument *volume* is a numeric expression in the range 0 to 15. If you omit this argument, the default value is 8. The *voice* argument is a numeric expression in the range 0 to 2. The default for voice is 0. A **SOUND ON** statement must be executed before these two arguments are included; otherwise, an "Illegal function call" error will result.

■ **Example 1**

This short fragment creates fifty random sounds whose frequency ranges from 37 to 5,000 hertz, and whose duration ranges from one clock tick to one second:

```
10 FOR I = 1 TO 50
20   FREQ = INT(4964*RND) + 37
30   DUR = INT(173*RND)/10 + 1
40     SOUND FREQ, DUR
50 NEXT
```

■ **Example 2**

This next program fragment produces a glissando up and down:

```
10 FOR I = 440 TO 1000 STEP 5
20    SOUND I, I/1000
30 NEXT
40 FOR I = 1000 TO 440 STEP -5
50    SOUND I, I/1000
60 NEXT
```

SPACE$ Function

■ Syntax

SPACE$(n)

■ Action

Returns a string of spaces of length n

■ Remarks

The expression n is rounded to an integer and must be in the range 0 to 255.

■ Example

```
10 FOR I=1 TO 5
20    X$=SPACE$(I)
30    PRINT X$;I
40 NEXT I
```

Output:

```
1
  2
   3
    4
     5
```

■ See Also

SPC

■ **Syntax**

SPC(*n*)

■ **Action**

Skips *n* spaces in a **PRINT** statement

■ **Remarks**

SPC may only be used with **PRINT** and **LPRINT** statements. The argument *n* must be in the range 0 to 255. A semicolon (;) is assumed to follow the **SPC(*n*)** command.

■ **Example**

```
PRINT "OVER" SPC(15) "THERE"
```

Output:

```
OVER           THERE
```

■ **See Also**

SPACE$

SQR Function

■ Syntax

SQR(*n*)

■ Action

Returns the square root of *n*

■ Remarks

The argument *n* must be $> = 0$.

■ Example

```
10 FOR X=10 TO 25 STEP 5
20    PRINT X, SQR(X)
30 NEXT X
```

Output:

```
10          3.162278
15          3.872984
20          4.472136
25          5
```

■ **Syntax**

STICK(*n*)

■ **Action**

Returns the *x* and *y* coordinates of the two joysticks

■ **Remarks**

The argument *n* is a numeric expression returning an unsigned integer in the range 0 to 3.

Table 6.11

Values returned by STICK

n =	Result
0	Returns the x coordinate for joystick A. Also stores the x and y values for both joysticks for the following function calls:
1	Returns the y coordinate of joystick A.
2	Returns the x coordinate of joystick B.
3	Returns the y coordinate of joystick B.

■ **Example**

This example displays the value of the *x,y* coordinate for joystick A:

```
30  CLS
40  LOCATE 1,1
50  PRINT "X=5";STICK(0)
60  PRINT "Y=5";STICK(1)
70  INPUT "Press RETURN to continue program"; DUMMY$
    .
    .
    .
```

STOP Statement

■ **Syntax**

STOP

■ **Action**

Terminates program execution and returns to command level

■ **Remarks**

STOP statements may be used anywhere in a program to terminate execution. **STOP** is often used for debugging. When a **STOP** is encountered, the following message is printed:

    Break in line *nnnnn*

With the interpreter, the **STOP** statement does not close files and returns to command level after the **STOP** is executed. Execution is resumed by issuing a **CONT** command.

■ **Example**

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
```

Output:

```
? 1,2,3
BREAK IN 30

PRINT L
 30.76923
CONT
 115.9
```

Note that because the **CONT** command is included here, this particular example works only with the interpreter.

■ Compiler/Interpreter Differences

With compiler, **STOP** closes all open files.

If the /**D**, /**E**, or /**X** compiler options are turned on, the **STOP** message prints the line number at which execution has stopped, as long as your program has line numbers.

STR$ Function

STR$(*n*)

■ **Action**

Returns a string representation of the value of *n*

■ **Example**

```
5   REM ARITHMETIC FOR KIDS
10  INPUT "TYPE A NUMBER";N
20  ON LEN(STR$(N))  GOSUB 30,100,200,300,400,500
.
.
.
```

■ **See Also**

VAL

■ **Syntax**

STRIG(n)

■ **Action**

Returns the status of a specified joystick trigger

■ **Remarks**

The numeric expression n is an unsigned integer in the range 0 to 3, designating which trigger is to be checked. In the **STRIG(n)** function, the values returned for n can be:

0 Returns -1 if trigger A was pressed since the last STRIG(0) statement; returns 0 if not.

1 Returns -1 if trigger A is currently down, 0 if not.

2 Returns -1 if trigger B was pressed since the last STRIG(2) statement, 0 if not.

3 Returns -1 if trigger B is currently down, 0 if not.

When a joystick event trap occurs, that occurrence of the event is destroyed. Therefore, the **STRIG(n)** function will always return false inside a subroutine, unless the event has been repeated since the trap. So if you wish to perform different procedures for various joysticks, you must set up a different subroutine for each joystick, rather than including all the procedures in a single subroutine.

■ **Example**

In this example an endless loop is created to beep whenever the trigger button on joystick 0 is pressed:

```
1O IF STRIG(O) THEN BEEP
20 GOTO 10
```

STRIG Function

■ **Compiler/Interpreter Differences**

See compiler note under "ON STRIG Statement."

STRIG ON, STRIG OFF, STRIG STOP Statements

■ **Syntax**

STRIG ON
STRIG OFF
STRIG STOP

■ **Action**

The **STRIG ON** statement enables event trapping of joystick activity.

The **STRIG OFF** statement disables event trapping of joystick activity.

The **STRIG STOP** statement disables event trapping of joystick activity.

■ **Remarks**

The **STRIG ON** statement enables joystick event trapping by an **ON STRIG** statement (see "STRIG Statement"). While trappping is enabled, and if a non-zero line number is specified in the **ON STRIG** statement, BASIC checks between every statement to see if the joystick trigger has been pressed.

The **STRIG OFF** statement disables event trapping. If a subsequent event occurs (i.e., if the trigger is pressed), it will not be remembered when the next **STRIG ON** is invoked.

The **STRIG STOP** statement disables event trapping, but if an event occurs it will be remembered, and the event trap will take place as soon as trapping is reenabled.

■ **See Also**

STRIG

STRING$ Function

■ Syntax

STRING$$(m, n)$
STRING$$(m, x\$)$

■ Action

Returns a string of length m whose characters all have ASCII code n or whose characters are all the first character of $x\$$.

■ Example 1

```
10 DASH$ = STRING$(10,45)
20 PRINT DASH$;"MONTHLY REPORT";DASH$
```

Output:

```
----------MONTHLY REPORT----------
```

■ Example 2

```
10 A$ = "HOUSTON"
20 X$ = STRING$(8,A$)
30 PRINT X$
```

Output:

```
HHHHHHHH
```

■ Syntax

SWAP *variable, variable*

■ Action

Exchanges the values of two variables

■ Remarks.

Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

If the second variable is not already defined when **SWAP** is executed, an "Illegal function call" error will result.

■ Example

```
10 A$=" ONE  "  :  B$=" ALL  "  :  C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
```

Output:

```
ONE FOR ALL
ALL FOR ONE
```

SYSTEM Command

■ **Syntax**

SYSTEM

■ **Action**

Closes all open files and returns control to the operating system

■ **Remarks**

When a **SYSTEM** command is executed, all files are closed, and BASIC performs an exit to the operating system.

■ **Syntax**

TAB(*col*)

■ **Action**

Moves the print position to *col*

■ **Remarks**

If the current print position is already beyond *col* **TAB** goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. The argument *col* must be in the range 1 to 255. **TAB** can be used only in **PRINT** and **LPRINT** statements.

■ **Example**

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
```

Output:

```
NAME                    AMOUNT

G. T. JONES             $25.00
```

TAN Function

■ **Syntax**

TAN(x)

■ **Action**

Returns the tangent of x, where x is in radians

■ **Remarks**

With the interpreter, if **TAN** overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

■ **Example**

```
10 Y=Q*TAN (X) /2
```

■ **Syntax**

TIME$

■ **Action**

Retrieves the current time

■ **Remarks**

The **TIME$** function returns an eight-character string in the form
hh:mm:ss, where *hh* is the hour (00 through 23), *mm* is minutes (00 through
59), and *ss* is seconds (00 through 59). A 24-hour clock is used; 8:00 p.m.,
therefore, is shown as 20:00:00.

To set the time, use the **TIME$** statement.

■ **Example**

```
10 PRINT TIME$
```

Sample output:

```
00:23:45
```

■ **See Also**

TIME$ Statement

TIME$ Statement

■ **Syntax**

TIME$ =*stringexpression*

■ **Action**

Sets the time

■ **Remarks**

The *stringexpression* must be in one of the following forms:

   hh   (sets the hour; minutes and seconds default to 00)

   hh:mm (sets the hour and minutes; seconds default to 00)

   hh:mm:ss
        (sets the hour, minutes, and seconds)

A 24-hour clock is used; 8:00 p.m., therefore, would be entered as 20:00:00.

This statement complements the **TIME$** function, which retrieves the time.

■ **See Also**

TIME$ Function

■ **Example**

The current time is set at 8:00 a.m:

```
10 TIME$="08:00:00"
```

■ **Syntax**

TIMER

■ **Action**

Returns the number of seconds elapsed since midnight

■ **Remarks**

The timer function can be used with the **RANDOMIZE** statment to generate a random number. It can also be used to time programs or parts of programs.

■ **Example**

The following program searches for the prime numbers from 3 to 10000. The **TIMER** function is used to time the execution speed of this program.

```
DEFINT A-Z
DIM MARK(10000)
START! = TIMER
NUM = O
FOR N = 3 TO 10000 STEP 2
    IF NOT MARK(N) THEN
        'PRINT N,         'To print the primes, remove the remark
                          'delimiter in front of the PRINT statement
        DELTA = 2*N
        FOR I = 3*N TO 10000 STEP DELTA
            MARK(I) = -1
        NEXT
        NUM = NUM + 1
    END IF
NEXT
FINISH! = TIMER
PRINT
PRINT "Program took" FINISH!-START! "seconds to find the" NUM "primes"
END
```

Output:

```
Program took 2.632813 seconds to find the 1228 primes
```

TIMER ON, TIMER OFF, TIMER STOP Statements

■ **Syntax**

TIMER ON
TIMER OFF
TIMER STOP

■ **Action**

TIMER ON enables event trapping during real time. **TIMER OFF** disables event trapping during real time. **TIMER STOP** suspends real time event trapping.

■ **Remarks**

The **TIMER ON** statement enables real time event trapping by an **ON TIMER** statement While trapping is enabled with the **ON TIMER** statement, BASIC checks between every statement to see if the timer has reached the specified level. If it has, the **ON TIMER** statement is executed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent **TIMER ON** is used.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an **ON TIMER** statement will be executed as soon as trapping is enabled.

■ **See Also**

ON TIMER

■ **Syntax**

TRON
TROFF

■ **Action**

Traces the execution of program statements

■ **Remarks**

As an aid in debugging, the **TRON** statement may be executed in either direct or indirect mode. With **TRON** in operation, each line number of the program is printed on the screen as it is executed.

The numbers appear enclosed in square brackets. The trace flag is disabled with the **TROFF** statement (or when a **NEW** command is executed).

■ **Example**

```
TRON

10 K=10
20 FOR J=1 TO 2
30    L=K + 10
40    PRINT J;K;L
50    K=K + 10
60 NEXT J
70 END
```

Output:

```
[10] [20] [30] [40]  1   10   20
[50] [60] [30] [40]  2   20   30
[50] [60] [70]
```

Note that this example is for the interpreter only.

TRON/TROFF Statements

■ **Compiler/Interpreter Differences**

To use **TRON/TROFF**, the compiler debug option **/D** must be turned on. Otherwise, **TRON** and **TROFF** are ignored, and a warning message is generated.

■ **Syntax**

UNLOCK [[#]] *filenum* [[,{ *record*[[*start*]] TO *end*}]]

■ **Action**

Releases access restrictions applied to specified portions of a file

■ **Remarks**

The **UNLOCK** statement should be used only after a **LOCK** statement. See "LOCK Statement" for examples and a complete discussion.

■ **See Also**

LOCK

USR Function

■ **Syntax**

USR[*digit*][(*argument*)]

■ **Action**

Calls an assembly language subroutine

■ **Remarks**

The argument *digit* specifies which **USR** routine is being called. See the
"DEF USR Statement," for the rules governing *digit*. If *digit* is omitted,
USR0 is assumed.

The *argument* is the value passed to the subroutine. It may be any numeric
or string expression.

If a segment other than the default segment (data segment) is to be used, a
DEF SEG statement must be executed prior to a **USR** function call. The
address given in the **DEF SEG** statement determines the segment address
of the subroutine.

For each **USR** function, a corresponding **DEF USR** statement must be
executed to define the **USR** call offset. This offset and the currently active
DEF SEG segment address determine the starting address of the subrou-
tine.

■ **See Also**

DEF USR

■ **Example**

```
100 DEF SEG=_&H8000
110 DEF USR0=0
120 X=5
130 Y = USR0 (X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the value must be
consistent with the argument passed.

■ **Compiler/Interpreter Differences**

The **USR** function is implemented in the compiler to call machine language subroutines. However, there is no way to pass parameters with the **USR** function, except by using **POKE** statements to protected memory locations that are later accessed by the machine language routine.

If this method is used, the **USR** function must preserve the values of all registers except **BX**. The **USR** function must return the integer result in the **BX** register. There are two alternatives to using **POKE** statements to pass parameters:

1. If the machine language routine is short enough, it can be stored by making a string containing the ASCII values corresponding to the hexadecimal values of the routine. Use the **CHR$** function to insert ASCII values in the string. The start of the routine can then be found by using the **VARPTR** function. For example, for the string A$, VARPTR (A$) will return the address of the low byte of the string length. The next address contains the high byte of the string length. The next two addresses are: first, the least significant byte, and second, the most significant byte, of the actual address of the string.

    This setup of the string space differs from that of the interpreter. Thus, to find the actual starting address of the routine, use the following instructions:

    ```
    10 A$ = "String containing routine"
    20 I% = VARPTR(A$)
    30 AD = PEEK(I% + 3) * 256 + PEEK(I% + 2)
    40 REM  AD is the start address
    50 REM  of the routine
    ```

    String contents move around in the string space, so any absolute references must be adjusted to reflect the current memory location of the routine.

2. A better alternative is to use an assembler to assemble your subroutines. Then the subroutines can be linked directly to the compiled program and referenced using the **CALL** statement.

VAL Function

VAL(*string*)

■ **Action**

Returns the numeric value of string *string*

■ **Remarks**

The *string* must be a numeric character stored as a string. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

    VAL("   -3")

returns -3.

See the "STR$ Function," for details on numeric-to-string conversion.

■ **See Also**

STR$

■ **Example**

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN _
      PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN _
      PRINT NAME$ TAB(25) "LONG BEACH"
   .
   .
   .
```

■ **Syntax 1**

VARPTR(*variablename*)

■ **Syntax 2**

VARPTR(# *filenumber*)

■ **Action**

Syntax 1:

Returns the address of the first byte of data identified with *variablename*.
The *variablename* must have been defined prior to the execution of the
VARPTR function. Otherwise an "Illegal function call" error results.
Variables are defined by executing any reference to the variable.

Any type variable name may be used (numeric, string, array). For string
variables, the address of the first byte of the string descriptor is returned.
The address returned will be an integer in the range 32767 to -32768. If a
negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so
that it may be passed to an assembly language subroutine. A function call
of the form

VARPTR (A (0))

is usually specified when passing an array, so that the lowest-addressed ele-
ment of the array A in the above case, is returned.

Note

> All simple variables should be assigned before calling **VARPTR** for an
> array, because the addresses of the arrays change whenever a new sim-
> ple variable is assigned.

Syntax 2:

VARPTR Function

For sequential files, returns the starting address of the disk I/O buffer assigned to *filenumber*. For random files, returns the address of the FIELD buffer assigned to *filenumber*.

■ **Example**

100 X=USR (VARPTR (Y))

■ **Syntax**

VARPTR$(*variablename*)

■ **Action**

Returns a character form of the memory address of the variable in a form that is compatible for programs that may later be compiled

■ **Remarks**

The *variablename* is the name of a variable in the program.

VARPTR$ is often used to execute substrings with the **DRAW** and **PLAY** statements in programs that will later be compiled. With programs that will not be later compiled, the standard syntax of the **PLAY** and **DRAW** statements will be sufficient to produce desired effects.

The *variablename* must have been defined prior to the execution of the **VARPTR** function. Otherwise an "Illegal function call" error results. Variables are defined by executing any reference to the variable.

VARPTR$ returns a three-byte string in the form:

byte 0 = type
byte 1 = low byte of address
byte 2 = high byte of address

Note, however, that the individual parts of the string are not considered characters.

Note

Because array addresses, string addresses and file data blocks change whenever a new variable is assigned, it is unsafe to save the result of a **VARPTR** function in a variable. It is recommended that **VARPTR** be executed before each use of the result.

VARPTR$ Function

■ **Example**

This uses the **PLAY** subcommand **X** (execute), plus the contents of A$, as the string expression in the **PLAY** statement:

```
10 PLAY "X"+VARPTR$(A$)
```

■ **Syntax**

VIEW [[SCREEN] [(*x1,y1*)-(*x2,y2*) [,[*color*] [,[*border*]]]]]

■ **Action**

Defines screen limits for graphics activity

■ **Remarks**

VIEW defines a "physical viewport" limit from *x1,y1* (upper left *x,y* coordinates) to *x2,y2* (lower right *x,y* coordinates). The *x* and *y* coordinates must be within the physical bounds of the screen. The physical viewport defines the rectangle within the screen into which graphics may be mapped.

RUN, and SCREEN, and VIEW with no arguments, define the entire screen as the viewport.

The *color* attribute allows the user to fill the view area with a color. If *color* is omitted, the view area is not filled.

The *border* attribute allows the user to draw a line surrounding the viewport if space for a border is available. If *border* is omitted, no border is drawn.

The SCREEN option dictates that the *x* and *y* coordinates are absolute to the screen, not relative to the border of the physical viewport, and only graphics within the viewport will be plotted.

■ **Examples**

All points plotted are relative to the viewport (that is, *x1* and *y1* are added to the *x* and *y* coordinates before putting the point down on the screen):

VIEW (*x1,y1*)-(*x2,y2*)

After the following statement is executed, the point set down by the statement

PSET (0,0),3

would actually be at the physical screen location 10,10:

VIEW Statement

VIEW (10,10) - (200,100)

All coordinates are screen absolute rather than viewport relative.

After the statement is executed in the following example, the point set down by the statement

PSET (0,0),3

would actually not appear because 0,0 is outside of the viewport. The statement

PSET (10,10),3

is within the viewport, and places the point in the upper-left hand corner of the viewport):

VIEW SCREEN (10,10) - (200,100)

A number of **VIEW** statements can be executed. If the newly described viewport is not wholly within the previous viewport, the screen can be re-initialized with the **VIEW** statement. Then the new viewport may be stated. If the new viewport is entirely within the previous one, as in the following example, the intermediate **VIEW** statement is not necessary. This example opens three viewports, each smaller than the previous one. In each case, a line that is defined to go beyond the borders is programmed, but appears only within the viewport border.

```
240 SCREEN 2
260 CLS
280 VIEW: REM ** Make the viewport the entire screen.
300 VIEW (10,10) - (300,180),,1
320    CLS
340    LINE (0,0) - (310,190),1
360    LOCATE 1,11: PRINT "A big viewport"
380 VIEW SCREEN (50,50)-(250,150),,1
400    CLS:REM** Note, CLS clears only viewport
420    LINE (300,0)-(0,199),1
440    LOCATE 9,9: PRINT "A medium viewport"
460 VIEW SCREEN (80,80)-(200,125),,1
480    CLS
500    CIRCLE (150,100),20,1
520    LOCATE 11,9: PRINT "A small viewport"
```

■ **Syntax**

VIEW PRINT [*topline* **TO** *bottomline*]

■ **Action**

Sets the boundaries of the screen text window

■ **Remarks**

VIEW PRINT without *topline* and *bottomline* parameters initializes the whole screen area as the text window.

Statements and functions which operate within the defined text window include **CLS, LOCATE, PRINT,** and **SCREEN.**

The screen editor will limit functions such as scroll and cursor movement to the text window.

■ **See Also**

VIEW

WAIT Statement

■ **Syntax**

WAIT *portnumber*,i[,j]

■ **Action**

Suspends program execution while monitoring the status of a machine input port

■ **Remarks**

The arguments *i* and *j* are integer expressions.

The **WAIT** statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression *j*, and then AND'ed with *i*. If the result is zero, GW-BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *j* is omitted, it is assumed to be zero

---

Warning

It is possible to enter an infinite loop with the **WAIT** statement, in which case it will be necessary to manually restart the machine. To avoid this, **WAIT** must have the specified value at *portnumber* during some point in the program execution.

■ **Example**

100 WAIT 32,2

■ **Statement Syntax**

WHILE *condition*
.
.
.
WEND

■ **Action**

Executes a series of statements in a loop as long as a given condition is true

■ **Remarks**

If the *condition* is true (that is, if it does not equal zero), then *statements* are executed until the **WEND** statement is encountered. BASIC then returns to the **WHILE** statement and checks *condition*. If it is still true, the process is repeated. If it is not true (or if it equals zero), execution resumes with the statement following the **WEND** statement.

WHILE...WEND loops may be nested to any level. Each **WEND** will match the most recent **WHILE**. An unmatched **WHILE** statement causes a "WHILE without WEND" error message to be generated, and an unmatched **WEND** statement causes a "WEND without WHILE" error message to be generated.

Warning

Do not direct program flow into a **WHILE...WEND** loop without entering through the **WHILE** statement, as this will confuse structuring of control flow.

▩ Example 1

ᴛhe following fragment performs a bubble sort on the array a$. The
second line makes the variable "flips" true by assigning it a non-zero value;
this forces one pass through the **WHILE...WEND** loop. When there are
finally no more swaps, then all the elements of a$ are sorted, flips is false
(that is, equal to zero), and the program continues execution with the line
following wend:

```
rem **bubble sort array a$**
flips=1          'force one pass through loop
while flips
    flips=0
    for i=2 to ubound(a$)
        if a$(i-1)>a$(i) then gosub switch
    next i
wend
 .
 .
 .
end
switch:
    flips = 1
    swap a$(i-1),a$(i)
return
```

■ Example 2

The **WHILE...WEND** loop at the beginning of this program fragment calls a subroutine which prints the first five lines of the file chosen. If this is the correct file, the user answers "y", the **WHILE...WEND** loop is no longer true, and the program continues. If the user answers "n", the **WHILE...WEND** loop repeats.

```
r$ = "n"   'This forces first pass through while/wend loop.
while r$ = "n"
   cls
   input "Name of file to be updated:   ",f$
   gosub head
   input "Is this the correct file (y or n)";r$
wend
.
.
.
end
head:
   print "The first five line of ";f$;" are:" : print
   open f$ for input as #1
   for i = 1 to 5
       line input #1, temp$
       print temp$
   next
   close #1 : print
return
```

WIDTH Statement

■ **Syntax**

WIDTH [[LPRINT]] *size*

■ **Action**

Sets the printed line width in number of characters for the screen or the line printer

■ **Remarks**

If the **LPRINT** option is omitted the line width is set at the screen; if **LPRINT** is included, the line width is set at the line printer.

The *size* argument must be in the range 15 to 255. The default for *size* is 72. If *size* equals 255, the line width is "infinite"; that is, no carriage return is ever inserted. However, the position of the cursor or print head, as given by the **POS** or **LPOS** functions, returns to zero after position 255.

The **WIDTH** statement may cause the screen to clear.

■ **Compiler/Interpreter Differences**

The compiler implements an enhanced version of the interpreter's **WIDTH** statement in which files as well as devices can be assigned an output line width.
The possible syntaxes of this statement are explained in the following list:

Syntax	Action
WIDTH *size*	Sets the width of the terminal screen to *size*.
	The maximum width is 255.
WIDTH *filenumber,size*	Sets the width of a file opened to an output device (e.g., LST or CON) to *size*.
	The *filenumber* is the number associated with the file in the **OPEN** statement.
	This form permits altering the width while a file is open, since the statement takes place immediately.

WIDTH *device,size*   Sets the width of *device* (a device filename) to *size*.

The *device* should be a string value enclosed in double quotes; for example:

"CONS:"

Note that this width assignment is *deferred* until the next **OPEN** statement affecting the device; the assignment does not affect output for an already open file.

■ **Example 1**

In the following example, the width of the lineprinter is set to 120 columns:

```
width "lpt1:",120
```

■ **Example 2**

In the following example, the record width in file #1 is set to 40 columns:

```
width #1,40
```

■ **Example 3**

In the following example, file #1 is the console display:

```
open "scrn:" for output as #1
test$ = "1234567890"
width #1,2
   print #1, test$         °
width #1,4
   locate 7,1
   print #1, test$
```

Output:

```
12
34
56
78
90

1234
```

5678
90

■ **Syntax**

WINDOW [[[SCREEN]] $(x1,y1)$-$(x2,y2)$]]

■ **Action**

Defines the logical dimensions of the current viewport

■ **Remarks**

The clause contains the world coordinates specified by the programmer to define the coordinates of the lower left and upper right screen border.

SCREEN inverts the y axis of the world coordinates so that screen coordinates coincide with the traditional Cartesian arrangement: x increases left to right, and y decreases top to bottom.

WINDOW allows the user to redefine the screen border coordinates.

WINDOW allows the user to draw lines, graphs, or objects in space not bounded by the physical dimensions of the screen. This is done by using programmer-defined coordinates called "world coordinates." When the programmer has redefined the screen, graphics can be drawn within a customized mapping system.

BASIC converts world coordinates into physical coordinates for subsequent display within the current viewport. To make this transformation from world space to the physical space of the viewing surface (screen), one must know what portion of the (floating point) world coordinate space contains the information to be displayed. This rectangular region in world coordinate space is called a Window.

RUN, or **WINDOW** with no arguments, disables "window" transformation.

The **WINDOW SCREEN** variant inverts the normal Cartesian direction of the y coordinate. Consider the following:

WINDOW Statement

In the default, a section of the screen appears as:

```
-------------------------------------------------
| 0,0                  50,0                100,0 |
|          ||                                    |
|          ||                                    |
|          \/  y increases                       |
|                      100,0                      |
|                                                |
|                                                |
|                                                |
| 0,100                50,100               100,100 |
-------------------------------------------------
```
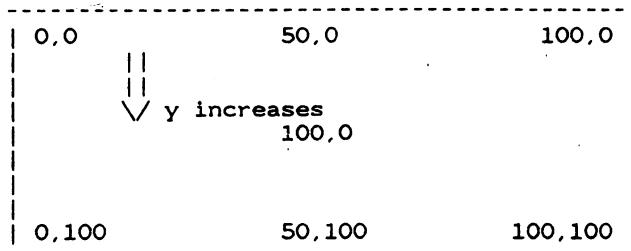
now execute:

`WINDOW (-1,-1)-(1,1)`

and the screen appears as:

```
-------------------------------------------------
| -1,1                 0,1                   1,1 |
|          /e  y increases                       |
|          ||                                    |
|          ||                                    |
|                      0,0                       |
|          ||                                    |
|          ||                                    |
|          \/  y decreases                       |
| -1,-1                0,-1                  1,-1 |
-------------------------------------------------
```

866
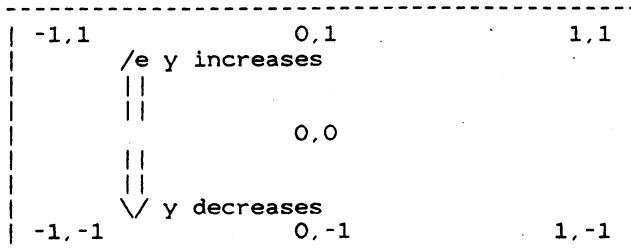
If the variant:

```
WINDOW SCREEN  (-1,-1)-(1,1)
```

is executed then the screen appears as:

```
-------------------------------------------
| -1,-1              0,-1           1,-1 |
|           /\ y decreases                |
|           ||                            |
|.          ||                            |
|                    0,0                  |
|                                         |
|           ||                            |
|           ||                            |
|           \/ y increases                |
| -1,1               0,1            1,1 |
-------------------------------------------
```
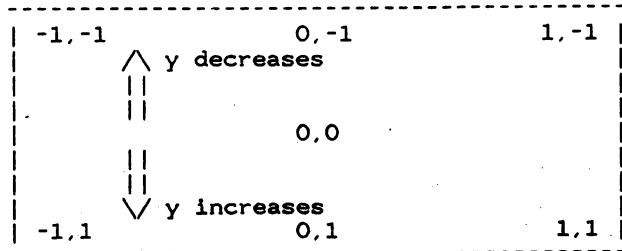
The following example illustrates two lines with the same endpoint coordinates. The first is drawn on the default screen, and the second is on a redefined window.

```
190 SCREEN 2
200 LINE (100,100) - (150,150), 1
220 LOCATE 2,20:PRINT "The line on the default screen"
240 WINDOW SCREEN (100,100) - (200,200)
260 LINE (100,100) - (150,150), 1
280 LOCATE 8,18:PRINT"& the same line on a redefined window"
```

WRITE Statement

■ **Syntax**

WRITE[[*expressionlist*]]

■ **Action**

Outputs data to the screen

■ **Remarks**

If *expressionlist* is omitted, a blank line is output. If *expressionlist* is included, the values of the expressions are output to the screen. The expressions in the list may be numeric and/or string expressions. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, GW-BASIC inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the **PRINT** statement.

■ **See Also**

PRINT

■ **Example**

This example shows the difference between the **PRINT** and **WRITE** statements:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
30 PRINT A,B,C$
```

Output:

```
80, 90,"THAT'S ALL"
  80      90        THAT'S ALL
```

■ **Syntax**

WRITE# *filenumber,expressionlist*

■ **Action**

Writes data to a sequential file

■ **Remarks**

The *filenumber* is the number under which the file was opened for **OUT-PUT** or **APPEND** in the **OPEN** statement. The expressions in the *expressionlist* are string or numeric expressions. They must be separated by commas.

The difference between **WRITE#** and **PRINT#** is that **WRITE#** inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. A newline is inserted, once the last item in the list has been written to the file.

If a **WRITE#** statement attempts to write data to a sequential file to which access has been restricted by a **LOCK** statement, two options are available. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error handling routines can trap and examine this error. If error trapping is not active the error message is:

Permission denied

■ **See Also**

OPEN, LOCK, PRINT#, WRITE

■ · **Examples**

These two short programs, and their output, illustrate the difference between **WRITE#** and **PRINT#** statements:

```
10 A$="TELEVISION, COLOR":B$="$599.00"
20 OPEN "PRICES" FOR OUTPUT AS 1
30 PRINT #1,A$,B$
```

WRITE# Statement

```
40 CLOSE #1
50 OPEN "PRICES" FOR INPUT AS 1
60 INPUT #1,A$,B$
70 PRINT A$;TAB(25);B$
```

Output:

TELEVISION   COLOR $599.00

Substituting the following line

```
30 WRITE #1,A$,B$
```

in the previous program gives this output:

TELEVISION,COLOR $599.00

◊

Appendix A
ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	033	21H	!
001	01H	SOH	034	22H	"
002	02H	STX	035	23H	#
003	03H	ETX	036	24H	$
004	04H	EOT	037	25H	%
005	05H	ENQ	038	26H	&
006	06H	ACK	039	27H	'
007	07H	BEL	040	28H	(
008	08H	BS	041	29H	)
009	09H	HT	042	2AH	*
010	0AH	LF	043	2BH	+
011	0BH	VT	044	2CH	,
012	0CH	FF	045	2DH	-
013	0DH	CR	046	2EH	.
014	0EH	SO	047	2FH	/
015	0FH	SI	048	30H	0
016	10H	DLE	049	31H	1
017	11H	DC1	050	32H	2
018	12H	DC2	051	33H	3
019	13H	DC3	052	34H	4
020	14H	DC4	053	35H	5
021	15H	NAK	054	36H	6
022	16H	SYN	055	37H	7
023	17H	ETB	056	38H	8
024	18H	CAN	057	39H	9
025	19H	EM	058	3AH	:
026	1AH	SUB	059	3BH	;
027	1BH	ESCAPE	060	3CH	<
028	1CH	FS	061	3DH	=
029	1DH	GS	062	3EH	>
030	1EH	RS	063	3FH	?
031	1FH	US	064	40H	@
032	20H	SPACE			

Dec=decimal, Hex=hexadecimal (H), CHR=character. LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Dec	Hex	CHR		Dec	Hex	CHR
065	41H	A		097	61H	a
066	42H	B		098	62H	b
067	43H	C		099	63H	c
068	44H	D		100	64H	d
069	45H	E		101	65H	e
070	46H	F		102	66H	f
071	47H	G		103	67H	g
072	48H	H		104	68H	h
073	49H	I		105	69H	i
074	4AH	J		106	6AH	j
075	4BH	K		107	6BH	k
076	4CH	L		108	6CH	l
077	4DH	M		109	6DH	m
078	4EH	N		110	6EH	n
079	4FH	O		111	6FH	o
080	50H	P		112	70H	p
081	51H	Q		113	71H	q
082	52H	R		114	72H	r
083	53H	S		115	73H	s
084	54H	T		116	74H	t
085	55H	U		117	75H	u
086	56H	V		118	76H	v
087	57H	W		119	77H	w
088	58H	X		120	78H	x
089	59H	Y		121	79H	y
090	5AH	Z		122	7AH	z
091	5BH	[		123	7BH	{
092	5CH	\		124	7CH		
093	5DH	]		125	7DH	}
094	5EH	^		126	7EH	~
095	5FH	_		128	7FH	DEL
096	60H	`				

Dec=decimal, Hex=hexadecimal (H), CHR=character. LF=Line Feed,
FF=Form Feed, CR=Carriage Return, DEL=Rubout

Appendix B

Error Codes and Error Messages

Number	Message
1	NEXT without FOR

A variable in a **NEXT** statement does not correspond to any previously executed, unmatched **FOR** statement variable.

2	Syntax error

A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).

With BASIC, the incorrect line will be part of a **DATA** statement.

The interpreter automatically enters edit mode at the line that caused the error.

3	Return without GOSUB

A **RETURN** statement is encountered for which there is no previous, unmatched **GOSUB** statement.

4	Out of data

A **READ** statement is executed when there are no **DATA** statements with unread data remaining in the program.

5	Illegal function call

A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of:

1. A negative or unreasonably large subscript.
2. A negative or zero argument with **LOG**.
3. A negative argument to **SQR**.

4. A negative mantissa with a noninteger exponent.

5. A call to a **USR** function for which the starting address has not yet been given.

6. An improper argument to **MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR,** or **ON...GOTO.**

7. A negative record number used with **GET** or **PUT.**

6   `Overflow`

The result of a calculation is too large to be represented in the Microsoft BASIC number format. If underflow occurs, the result is zero and execution continues without an error.

7   `Out of memory`

A program is too large, or has too many **FOR** loops or **GOSUB** statements, too many variables, or expressions that are too complicated for a file buffer to be allocated.

8   `Undefined line`

A nonexistent line is referenced in a **GOTO, GOSUB, IF...THEN...ELSE,** or **DELETE** statement.

9   `Subscript out of range`

An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.

10   `Duplicate definition`

Two **DIM** statements are given for the same array; or, a **DIM** statement is given for an array after the default dimension of 10 has been established for that array.

11   `Division by zero`

A division by zero is encountered in an expression; or, the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12   Illegal direct

A statement that is illegal in direct mode is entered as a
direct mode command.

13   Type mismatch

A string variable name is assigned a numeric value or vice
versa; a function that expects a numeric argument is given a
string argument or vice versa.

14   Out of string space

String variables have caused BASIC to exceed the amount of
free memory remaining. Microsoft BASIC will allocate
string space dynamically, until it runs out of memory.

15   String too long

An attempt is made to create a string more than 255 charac-
ters long.

16   String formula too complex

A string expression is too long or too complex. The expres-
sion should be broken into smaller expressions.

17   Can't continue

An attempt is made to continue a program that:

1. Has halted due to an error.

2. Has been modified during a break in execution.

3. Does not exist.

18   Undefined user function

A USR function is called before the function definition
(DEF statement) is given.

19   No RESUME

An error handling routine is entered but contains no
RESUME statement.

20   RESUME without error

            A **RESUME** statement is encountered before an error han-
            dling routine is entered.

21   Unprintable error

            An error message is not available for the error condition
            that exists.

22   Missing operand

            An expression contains an operator with no operand follow-
            ing it.

23   Line buffer overflow

            An attempt has been made to input a line that has too
            many characters.

24   Device timeout

            The device you have specified is not available at this time.

25   Device fault

            An incorrect device designation has been entered.

26   FOR without NEXT

            A **FOR** statement was encountered without a matching
            NEXT.

27   Out of paper

            The printer device is out of paper.

28   Unprintable error

            An error message is not available for the condition which
            exists.

29   WHILE without WEND

            A **WHILE** statement does not have a matching **WEND**.

30   WEND without WHILE

            A **WEND** statement was encountered without a matching
            WHILE.

31-49   Unprintable error

An error message is not available for the condition which exists.

Disk Errors

50   Field overflow

A **FIELD** statement is attempting to allocate more bytes than were specified for the record length of a random file.

51   Internal error

An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared.

52   Bad file number

A statement or command references a file with a file number that is not **OPEN** or is out of the range of file numbers specified at initialization.

53   File not found

A **LOAD, KILL, NAME,** or **OPEN** statement/command references a file that does not exist on the current disk.

54   Bad file mode

An attempt is made to use **PUT, GET,** or **LOF** with a sequential file, to **LOAD** a random file, or to execute an **OPEN** statement with a file mode other than **I, O,** or **R.**

55   File already open

A sequential output mode **OPEN** statement is issued for a file that is already open; or a **KILL** statement is given for a file that is open.

56   Unprintable error

An error message is not available for the condition that exists.

57   Device I/O error

An I/O error occurred on a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the

error.

58   File already exists

The file name specified in a **NAME** statement is identical to a file name already in use on the disk.

59-60   Unprintable error

An error message is not available for the condition that exists.

61   Disk full

All disk storage space is in use.

62   Input past end

An **INPUT** statement is executed after all the data in the file has been **INPUT**, or for a null (empty) file. To avoid this error, use the **EOF** function to detect the end-of-file.

63   Bad record number

In a **PUT** or **GET** statement, the record number is either greater than the maximum allowed (32,767) or equal to zero.

64   Bad file name

An illegal form is used for the file name with a **LOAD**, **SAVE**, **KILL**, or **OPEN** statement (e.g., a file name with too many characters).

65   Unprintable error

An error message is not available for the condition that exists.

66   Direct statement in file

A direct statement is encountered while loading an ASCII-format file. The **LOAD** is terminated.

67   Too many files

An attempt is made to create a new file (using **SAVE** or **OPEN**) when all 255 directory entries are full.

68   Device Unavailable

The device that has been specified is not available at this time.

69   Communications buffer overflow

Not enough space has been reserved for communications I/O.

70   Disk write protected

The disk is write protected or is a disk that cannot be written to.

71   Disk not ready

Can be caused by a number of problems. The most likely is that the disk is not inserted properly.

72   Disk media error

A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk may be damaged or the disk drive may not be working properly.

74   Rename across disks

An attempt was made to rename a file with a new drive designation. This is not allowed.

75   Path/file access error

During an **OPEN, MKDIR, CHDIR,** or **RMDIR** operation, MS-DOS was unable to make a correct Path to File name connection. The operation is not completed.

76   Path not Found

During an **OPEN, MKDIR, CHDIR,** or **RMDIR** operation, MS-DOS was unable to find the path specified. The operation is not completed.

**   You cannot run BASIC as a Child of BASIC

No error number. During initialization, BASIC discovers that it is being run as a child. BASIC is not run and control returns to the parent copy of BASIC.

★ ★   Can't continue after SHELL

No error number. Upon returning from a child process, the
SHELL statement discovers that there is not enough
memory for BASIC to continue. BASIC closes any open files
and exits to MS-DOS.

Appendix C

Mathematical Functions
Not Intrinsic to BASIC

Derived Functions

Functions that are not intrinsic to Microsoft BASIC may be calculated as
follows.

Function   **Microsoft BASIC Equivalent**

SECANT   SEC(X)=1/COS(X)
COSECANT   CSC(X)=1/SIN(X)
COTANGENT   COT(X)=1/TAN(X)
INVERSE SINE   ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE   ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT   ARCSEC(X)=ATN(X/SQR(X*X-1))
                                      +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT   ARCCSC(X)=ATN(X/SQR(X*X-1))
                                      +(SGN(X)-1)*1.5708
INVERSE COTANGENT   ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE   SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE   COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT   TANH(X)=(EXP(X)-EXP(-X))/
                                      (EXP(X)+EXP(-X))
HYPERBOLIC SECANT   SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT   CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT   COTH(X)=(EXP(X)+EXP(-X))/
                                      (EXP(X)-EXP(-X))

INVERSE HYPERBOLIC
SINE   ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC
COSINE   ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC
TANGENT   ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC
SECANT   ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC
COSECANT   ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC
COTANGENT   ARCCOTH(X)=LOG((X+1)/(X-1))/2

Microsoft GW-BASIC Interpreter

Appendix D
OEM Manual Adaptation

This manual, the Microsoft GW-BASIC Interpreter manual, accompanies the Microsoft GW-BASIC Interpreter. It cannot be used in its entirety as a reference manual without adaptation to take advantage of the particular machine characteristics of a given OEM machine. This appendix identifies those portions of the manual that need to be adapted.

A thorough examination of all materials will enhance the relevance of the documentation to your specific implementation of GW-BASIC. With this in mind, we have addressed several key areas below. Note that with this release an IBM BASICA compatible version of the software is available and documentation changes so noted in this appendix.

D.1   General Issues

D.1.1   Event Trapping Statements

Not all implementations of Microsoft GW-BASIC support all of the events described. If an event trapping statement is encountered for an event that is not supported, a "Device unavailable" error will result.

D.1.2   Active and Visual Pages

The handling of active and visual pages is a machine-specific operation. You should consider expanding on your unit's approach in this section.

D.2   Language Reference Differences

Several elements of the BASIC language may be different or not included in your implementation. Some of these statements are listed below.

BEEP	ON KEY
CDBL$	OPEN
COLOR	OPEN COM
CSRLIN	PAINT
DRAW	PALETTE
ERDEV,ERDEV$	PEN
GET - Graphics	PLAY
INKEY$	PUT- Graphics
INPUT	POS

```
JIS$ - ECS        SCREEN
KEY               SOUND
KLEN - ECS         VAL
KPOS - ECS         VIEW
KTN$ - ECS        WINDOW
LOCATE
```

Read these sections closely and make the appropriate changes required for your specific implementation. A look at each statement and function alphabetically follows:

■ CIRCLE

The default value aspect ratio depends on particular screen hardware.

▣ CLEAR

In some versions, **CLEAR** supports the ability to further control video memory with the *videomemory* parameter. The following is the enhanced syntax for the compiler:

CLEAR [,[*location*][,[*stack*][,*videomemory*]]]

The *videomemory* parameter tells BASIC to ensure there is enough room for *videomemory* number of bytes for display memory. This allocates space for graphics modes that require large amounts of display memory, or for additional pages (each of which might be displayed on the screen). The *videomemory* parameter is not supported on most machines.

■ COLOR

Colors are display-hardware dependent.

■ LOCATE

Note that the meanings of the **LOCATE** argument, can vary according to your implementation. Specifically, the largest row number and column number will vary according to the particular machine. Also, whether the cursor blinks, and the speed at which it blinks, depends on the particular machine.

In addition, the start and stop options are not implemented on some machines. If they are available, the range will vary according to the particular machine.

■ **LPRINT**

Support for lprint depends on the lineprinters supported.

■ **NOISE**

Multiple voices require hardware support found only in certain machines. If you attempt to execute the **NOISE** statement on anything but a machine without this hardware support you will get an "Advanced feature error" message.

■ **PALETTE**

The range of colors for this statement depend on screen hardware.

■ **PCOPY**

If **PCOPY** is executed on a machine that does not support multiple screen pages, an "Advanced Feature Error" message results.

■ **PENON**

On some machines, the pen should not be used in the border area of the screen. Any values returned while the pen is in that area will be inaccurate.

■ **PLAY Statement**

The number of notes that can be played in the background at one time varies according to the particular machine. penon

■ PSET

The largest row number will vary according to the particular machine.

■ RND Function

The actual algorithm and random values for this function may vary with implementation.

■ SCREEN Statement

The options for this statement are display-hardware dependent.

■ SHELL

Some versions of BASIC will not allow the user to **SHELL** to another copy of BASIC. BASIC, when run as a child process, will recognize this before initialization and return to the parent copy of BASIC after issuing the message: "You cannot Shell to BASIC". This restriction is provided as an implementation option for cases where it is necessary to protect the integrity of the BASIC parent. BASIC will produce a child program when it uses the **SHELL** statement. It is not possible for BASIC to totally protect itself from its children.

When a **SHELL** statement is executed, many things may be going on. For example, files may be open and devices may be in use. The following guidelines will help to prevent child processes from harming the BASIC environment.

1. Hardware

    It is recommended that the state of all hardware be preserved during a **SHELL** command. The implementation interface provides a way for performing this task. However, it may be necessary to request that BASIC users refrain from using certain devices within child processes which are executed using the BASIC **SHELL** command. Specific areas of concern are:

    Screen Device
    > While useful information may be displayed by child processes, some children can damage existing screen mode parameters. This problem is further complicated by Screen Editor issues.

Interrupt Vectors

>Save and restore interrupt vectors the child intends to use. It seems reasonable to expect the child process to perform this task.

Other Hardware

>Many devices are placed in a specific state by BASIC. Such devices may include an Interrupt Controller, Counter Timers, DMA Controller, I/O Latch, and UARTS. These devices may be used by the child process without the user being aware of any limitations. This is a serious consideration which must be addressed with other GW-BASIC machine-specific considerations.

The File System

Warning

>Any child process which alters any open file in the BASIC parent shows poor programming technique, and may have unpredictable effects. If it is necessary to update such files, they should be closed in the parent before using SHELL, then re-opened upon return to the BASIC Parent.

Memory Management

a. Before BASIC "shells" to COMMAND, it will try to free any memory it is not then using, with one exception: when the BASIC interpreter is run with the /M: switch. When the /M: switch is set, BASIC assumes that the user intended to load something in the top of BASIC's memory block. This prevents BASIC from compressing the workspace before doing the SHELL. For this reason SHELL may fail with an "Out of memory" error when using the /M: switch.

A better method is to load machine language subroutines before BASIC is run. This can be accomplished by placing code at the end of machine language subroutines that allows them to exit to DOS and stay resident. For example:

```
CSEG    SEGMENT CODE    ;Machine language subroutine
RET                     ;Last instruction
START::
INT     27H             ;Terminate, stay resident
CSEG    ENDS
END     START
```

Be sure to "load" these subroutines before BASIC by running them. The AUTOEXEC.BAT file is very useful for this purpose.

b. A Child should *never* "Terminate and stay resident." Doing so may not leave BASIC enough room to expand its workspace to the original size. If BASIC cannot restore the workspace, all files are closed, the error message "SHELL can't continue" is printed, and BASIC exits to DOS.

Note

There is no restriction in the machine-independent portion of BASIC which prohibits GW-BASIC from running as a child of GW-BASIC. However, complications which arise from this configuration may lead to a decision by the OEM that this capability should not be implemented.

■ SOUND

Multiple voices require hardware support found only in certain machines. If you attempt to execute a program with multiple voices on a machine without such support, you will get an "Advanced feature error" message.

D.3   Extended Character Support

Extended character support (ECS) allows for use of GW-BASIC with the Kanji, Hanguel, and Chinese character sets. Double-byte characters occupy two character positions, and therefore the cursor and line wrap functions in the full screen editor differ slightly from those of some other editors. Normally, the cursor is between the current cursor marker and one position to

the left. With the full screen editor, the cursor is to the left of the entire character, whether the cursor marker is on the first or second position occupied by the character. This may affect the insert and delete modes slightly.

In addition, if you try to type a double-byte character at the last position on a physical line, the entire character is wrapped and a blank is placed at the last position on the line. The blank is not included in the logical line, however.

In addition to the above differences are the following changes to the GW-BASIC language.

D.3.1   Statement and Function Additions

This section documents additional statements and functions provided in addition to the standard GW-BASIC language to support national language character sets and other extended character capabilities.

CDBL$ Function

■ Syntax

CDBL$($x$$)   ◦

■ Action

Converts single-byte ASCII characters in the string $x$$ to their double-byte equivalents

■ Remarks

This function converts every single-byte ASCII character in $x$$ that is in the set {0-9, a-z, A-Z, plus (+), and minus (-)} to its double-byte equivalent character. The resulting string is one byte longer for every character that is converted.

If the result is longer than 255 bytes, a "String too long" error results.

■ Example 1

a	represents a single-byte ASCII character that has no equivalent double-byte character.
k	represents a single byte ASCII character that has an equivalent double-byte character.
K	represents a double-byte character.

■ Example 2

```
PRINT CDBL$ (akakKaa)
```

Output:

```
aKaKKaa
```

■ **Extended Character Support**

This function is supported only in double-byte versions of GW-BASIC.

CSNG$ Function

■ **Syntax**

CSNG$(*stringexpression*)

■ **Action**

Convert double-byte characters in *stringexpression* to their single-byte ASCII equivalents.

■ **Remarks**

This function converts every double-byte character in *stringexpression* that is in the set {0-9, a-z, A-Z, plus (+), and minus (-)} to its single-byte ASCII equivalent character. The resulting string is one byte shorter for every character that is converted.

■ **Example**

a	represents a single-byte ASCII character.
A	represents a double-byte character that has an equivalent single-byte ASCII character.
K	represents a double-byte character that has no equivalent single-byte ASCII character.

```
PRINT CSNG$ (aAaAKaa)
```

prints the following:

```
aaaaKaa
```

■ **Extended Character Support**

Only double-byte versions of GW-BASIC support this function.

■ Syntax

JIS$($x$$)$

■ Action

Converts the first character of the string expression to JIS representation

■ Remarks

The JIS$ function is used only with versions of GW-BASIC that provide double-byte character support.

If the first character of the string expression is a double-byte character, JIS$ returns a four-byte string containing the ASCII digits of the JIS code for the character.

If the first character of the string expression is not a double-byte character, JIS$ returns a three-byte string containing the ASCII digits of the ASCII code for the character.

■ Example

The following line returns a string containing the four ASCII digits 215F:

```
10 JIS$(CHR$(&H81)+CHR$(&7E))
```

The following line returns a string containing the three ASCII digits 041:

```
10 JIS$("A")
```

■ Extended Character Support

This function is supported only in double-byte versions.

KLEN Function

■ **Syntax**

KLEN($x\$$)

■ **Action**

Returns the number of characters in the specified string expression

■ **Remarks**

KLEN works exactly like **LEN**, except that it counts characters instead of bytes. Note that **LEN**($x\$$) minus **KLEN**($x\$$) is equal to the number of double-byte characters in $x\$$.

■ **Extended Character Support**

This function is supported only in double-byte versions of GW-BASIC.

■ **Syntax**

KPOS($x\$$,*character number*)

■ **Action**

Returns the byte number of *character number* in the string expression $x\$$.

■ **Remarks**

If the string expression contains fewer than *character number* characters, **KPOS** returns 0.

If the last byte of the string expression is the first byte of a double-byte character (in this case, the second byte is missing), that byte is ignored.

■ **Example**

The variable $x\$$ contains a string

"aKaKKaa"

where

"a"

represents an ASCII character and

"K"

is a two-byte MSKDC double-byte character equal to:

KPOS ($x\$$, 1) =1

(true for all $x\$$ except the null string)

```
KPOS (x$, 2) =2
KPOS (x$, 3) =4
KPOS (x$, 4) =5
KPOS (x$, 5) =7
KPOS (x$, 6) =9
KPOS (x$, 7) =10
KPOS (x$, 8) =0
```

KPOS Function

$KPOS(x\$,8)=0$ since the string contains only 7 characters.

■   **Extended Character Support**

Only double-byte versions of GW-BASIC support this function.

■ **Syntax**

KTN$ *(string)*

■ **Action**

Converts the first character of the specified string expression from
MSKDC to **KUTEN** representation

■ **Remarks**

If the first character of the string ' expression is a Kanji character, KTN$
returns a four-byte string containing the ASCII digits of the KUTEN code
for the character. For example

KTN$ (*"Kanji graphic for &K0132"*)

returns a string containing the four ASCII digits 0132.

If the first character of the string expression is not a Kanji character,
KTN$ returns a three-byte string containing the ASCII digits of the ASCII
code of the character. For example,

KTN$ ("A")

returns a string containing the three ASCII digits 065.

■ **Extended Character Support**

Only double-byte versions of GW-BASIC support this function.

D.3.2   Statement and Function Differences

The following pages describe language differences in the GW-BASIC
language arising out of implementation of extended character support.

■ ASC Function

For implementations that support double-byte characters, the **ASC** function behaves as follows: If the first character of the string is an MSKDC double-byte character, **ASC** returns the MSCDC representation of the character. If the first character of the string is not a double-byte character, **ASC** returns the code of the first byte of the string, just as it does in non-double-byte implementations of GW-BASIC. Note that double-byte characters are always two bytes long, so a string of length one will always be regarded as a non-double byte character, regardless of the code contained in it.

For example,

ASC ("ABC")

returns 65 (the ASCII code for A), since the first character of the string is not a double-byte character. Similarly,

ASC (CHR$ (&H21) +CHR$ (&H5F))

returns 318, the MSCDC representation of the double-byte character whose MSKDC representation is hexadecimal 215F.

■ CHR$ Function

For implementations that support double-byte characters, the syntax for this function is:

CHR$ (*n* [[,*n*...]])

You may use multiple arguments. If you specify an argument greater than 255, the MSCKC character is mapped to a double-byte MSKDC character. For example, CHR$(318) returns a double-byte string containing the hexadecimal codes 81 and 7E.

■ DATA Statement

DATA statements support double-byte characters.

■ INKEY$ Function

If double-byte characters are supported by the implementation, **INKEY$** returns a two-byte string containing the MSKDC representation for double-byte characters. For non-double-byte characters, **INKEY$** returns a one-byte string containing the ASCII code.

■ INPUT Statement

In implementations that support double-byte characters, **INPUT** variables can contain double-byte characters.

■ INPUT# Statement

Syntax

INPUT# *filenumber, variable1*[*, variable2*]...

Action

Reads data items from a sequential device or file and assigns them to program variables

Remarks

The *filenumber* is the number used when the file is opened for input. The variable list contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) Unlike **INPUT**, **INPUT#** does not print a question mark.

The data items in the file should appear just as they would if you were entering data in response to an **INPUT** statement. With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that is not a space, carriage return, or linefeed is as- sumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If GW-BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark

401

KTN$ Function

("), the string item will consist of all characters read between the first quotation mark and the second. This means a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or linefeed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example

```
INPUT 2,A,B,C
```

Extended Character Support

In implementations that support double-byte characters, the list of variables can contain double-byte characters.

■ INPUT$ Function

For implementations that support double-byte characters, you can use the following syntax:

INPUT*yen-sign*(*number-of-characters*, [#]*filenumber*)

This function performs the same action as **INPUT$** except that **INPUT***yen-sign* counts double-byte MSKDC double-byte characters as single characters. Therefore, the string that is returned will contain the specified number of characters but will contain more bytes if double-byte characters are encountered in the data. If the result is longer than 255 bytes, a "String too long" error results. A good way to avoid this error is to make *number-of-characters* less than or equal to 127.

■ PRINT USING Statement

For implementations that support double-byte characters, some of the characters differ from those used in other versions of BASIC. The differences are:

Table D.1

Double-Byte Character	Non-Double-Byte Character
&	\
(ã	&
yen yen	$ $
**yen	**$

- **SCREEN Function**

This function returns the MSCKC code for the character at the specified position, if that character is a double-byte character.

- **STRING$ Function**

For implementations that support double-byte characters, ASCII code n or the first character of x may specify a double-byte character. In this case, the string that is returned is twice as long as that returned for non-double-byte characters. For example,

```
STRING$(5,318)
```

returns a ten-byte string containing five copies of the double-byte character represented by MSCDC 318.

- **VAL Function**

For implementations that support double-byte characters, the **VAL** *string* argument may contain double-byte character codes.

D.4   Additional Statements

The following statements are non-standard statements found in some previous versions of BASIC. A particular implementation may support these statements and functions:

NOISE Statement

■ **Syntax**

NOISE *source, volume, duration*

■ **Action**

Generates noise through an external speaker

■ **Remarks**

The noise *source* is a numeric expression in the range 0 to 7. This parameter indicates gradations of noise from "periodic" (0 to 3) to "white" (4 to 7).

If you want to use a noise source of 3 or 7, the frequency of voice 3 is used instead of the system clock. You can set the voice 3 frequency with the **PLAY** or **SOUND** statements. The *volume* is a numeric expression in the range 0 to 15.

The *duration* is measured in clock ticks, with 18.2 ticks equal to one second. The *duration* must be a numeric expression in the range 0 to 65,535.

Sound produced by the **NOISE** statement goes to the external speaker. A **SOUND ON** statement must be executed before using **NOISE**, or an "Illegal function call" error will be generated.

■ **See Also**

PLAY, SOUND

■ **Example**

This example illustrates all the possible noise sources from 0 to 7. Note the use of three voices in the **PLAY** statement.

```
SOUND ON
FOR N = O TO 7
    NOISE N,15,250
    PLAY "","","VO"
    FOR I = 1 TO 6
        PLAY "","","V15;O="+VARPTR$(I)+";CDEF"
    NEXT I
```

NEXT N

D.5   IBM BASICA Compatability

The following changes are made such that the OEM can select at
configuration time whether his product is to be strictly IBM compatible, in
which case all of these changes take affect, or GW version-to-version com-
patible, in which case none of these changes take affect.

D.5.1   Screen/Keyboard I/O

D.5.1.1   Breaking from an INPUT statement

When IBM compatibility is selected, GW-BASIC will special- case Control-
C (^C) in an INPUT statement, and allow its use to break out of the state-
ment. ^C has no general effect on other statements.

IBM BASIC special-cases Control-C (^C) in an INPUT statement. Current
GW-BASIC does not. (break can be mapped to either or both ^C and ^-
Break; our example code uses Ctrl-Break).

D.5.1.2   INPUT # from keyboard

When IBM compatibility is selected, GW-BASIC will permit Ctrl-Z as a
character when reading from "KYBD:". Thus the program:

```
10 OPEN "KYBD:" FOR INPUT AS 1
20 INPUT #1,A$
30 GOTO 20
```

can be interrupted only by using Ctrl-Break.

Existing GW-BASIC treats CTRL-Z as EOF.

Note that IBM BASIC does not respond to CTRL-BREAK, but attempts to
accept it as a character. The result is that under IBM BASIC you cannot
exit the program. This effect is considered undesirable, and will not be sup-
ported.

D.5.1.3   Screen scroll

When IBM compatibility is selected, for non-disk I/O, GW- BASIC will output an extra $<CR><LF>$ if we are at the device width at the end of line output, and will mark the next field as the second field on the next line.

This program illustrates the current difference between IBM and GW BASIC:

```
10 CLS
20 LOCATE 10,1: PRINT "the line on which to print the X";
30 LOCATE 10,80 : PRINT "X"
```

When the IBM interpreter find that end-of-screen has been reached, it outputs an extra $<CR><LF>$ in addition to the one which follows the PRINT "X". Thus the output takes the form of the printed line, followed by a blank line, followed by "Ok". For existing GW-BASIC, there is no intervening blank line.

D.5.1.4   SCREEN Function in BASICA

When IBM compatibility is selected, GW-BASIC will make SCREEN (0,col) and SCREEN (row,0) legal and map them to SCREEN (1,col) and SCREEN (row,1) respectively. This protects user from breaking programs on 0, and provides reasonable results which probably reflect user intention.

The SCREEN function in IBM BASIC is broken in many respects:

1. SCREEN (0,col) is legal, and returns garbage.
2. SCREEN (row,0) also is legal. When the screen is cleared, it returns an ASCII space, except on row 22, where it returns garbage.
3. SCREEN (0,0) is not legal.
4. For SCREEN (25,col), when the screen is cleared, it returns '0' for all positions on row 25. If 24 is substituted for 25, values returned are 32 (ASCII space) for all column positions.

GW-BASIC currently handles all legal cases correctly. SCREEN (0,col), SCREEN (row,0), and SCREEN (0,0) are all illegal, and it always returns '32' for a blank character.

D.5.1.5   WIDTH and SCRN:

When WIDTH is used on the "SCRN:" device, as in:

```
WIDTH "SCRN:",size
```

"size" will be limited to 40 or 80, when IBM compatibility is selected.

D.5.1.6   Embedded CR's and LF's

When compatibility is selected, GW BASIC will not attempt to "eat" carriage returns or line feeds embedded in printed strings. For example:

```
10 PRINT "AAAAA";CHR$(13);CHR$(10);"BBBBB"
```

will produce (for IBM Basic, and GW Basic with compatibility selected:

```
    AAAAA

    BBBBB
 .EI
```
Where as GW without compatibility selected (as current GW) produces:

```
    AAAAA
    BBBBB
```

D.5.1.7   POKE 0:41A

When the IBM compatible example low-level code is used, the address 0:41A will be supported as documented in the IBM BASIC manual, to "clear the BIOS keyboard buffer". The example given by IBM is:

```
DEF SEG = 0
POKE 1050,PEEK(1052)
 .EI
Current GW BASIC support has the side affect of disabling
keyboard trapping for the rest of the program. This will
also be corrected.
```

D.5.1.8   Screen Scrolling

When IBM compatibility is selected, screen scrolling will occur in situations where the current cursor is below the text window. For example:

```
10 KEY OFF
20 LOCATE 25,1 : PRINT "Sample"
30 GO TO 20
```

This will scroll lines 1 through 24 in IBM BASIC and GW when compatibility is selected. It will not in standard GW.

D.5.2   File & Other I/O

D.5.2.1   Error Numbers

The tables below illustrate the resulting actions of the GW BASIC interpreter for compatibility on reporting certain errors relating to file-handling statements.

Abbreviations used:

```
"File not found" = FNF
"Path not found" = PNF
"Bad file number" = BFNum
"Bad file name" = BFName
"Device I/O Error" = DevIO
"Path/File Access Error" = PFA
```

DEVICE LENGTH = 0

Example: for file which exists as

```
"c:20
```

Error:

```
files ":20
```

Statement	IBM BASIC	GW (IBM Compat)	GW (GW Compat)
FILES	FNF	FNF	BFName
KILL	PNF	PNF		
NAME	FNF	FNF		
CHDIR	PNF	PNF		

RMDIR	PNF	PNF	
MKDIR	PNF	PNF	
OPEN	BFNum	BFNum	BFName
LOAD	|	|	|
SAVE	|	|	|
MERGE	|	|	|
LIST	|	|	|

ILLEGAL DEVICE in pathname

Examples: for file which exists as

```
"c:20
```

Errors:

```
files "?:20     files "huh:20
```

Statement	IBM BASIC	GW (IBM Compat)	GW (GW Compat)
FILES	FNF	FNF	BFName
KILL	PNF	PNF	|
NAME	BFName	BFName	|
CHDIR	PNF	PNF	|
RMDIR	PNF	PNF	|
MKDIR	PNF	PNF	|
OPEN	BFNum	BFNum	BFName
LOAD	|	|	|
SAVE	|	|	|
MERGE	|	|	|
LIST	|	|	|

\<FILESPEC\> OR \<PATHNAME\> IS NULL STRING

Examples:

```
files ""
chdir ""
```

Statement	IBM BASIC	GW (IBM Compat)	GW (GW Compat)
FILES	BFName	BFName	Legal
KILL	|	|	FNF
NAME	|	|	FNF
CHDIR	|	|	PNF
RMDIR	|	|	PNF

MKDIR	￤	￤	PNE
OPEN	BFNum	BFNum	FNE
LOAD	￤	￤	￤
SAVE	￤	￤	￤
MERGE	￤	￤	￤
LIST	￤	￤	￤

WILDCARD in pathname

Examples: for file which exists as

"c:20

Errors:

```
files "c:20*      files "c:20
Statement IBM BASIC       GW (IBM Compat) GW (GW Compat)
--------- -------         ----------      -----------
FILES    FNE              FNE **          FNE
KILL     PNF              PNF             FNE
NAME     BFName           BFName          PNF
CHDIR    PNF              PNF             ￤
RMDIR    DevIO            DevIO           ￤
MKDIR    PNF              PNF             ￤

OPEN     BFName           BFName          PNF
LOAD     ￤               ￤              ￤
SAVE     ￤               ￤              ￤
MERGE    ￤               ￤              ￤
LIST     ￤               ￤              ￤
```

** When IBM BASIC encounters this error, it prints the current directory heading before "File not found." Neither COMPAQ BASIC nor IBM Compatibility GW-BASIC does (it was decided that this compatibility choice did not justify the amount of code required to support it).

EXTRA FIELD TERMINATOR ('.') in filename

Example: for file which exists as

"c:20

Error:

files "c:20

```
Statement IBM BASIC          GW (IBM Compat) GW (GW Compat)
--------- -------            ----------      ----------
FILES     FNF                 FNF             FNF
KILL      PNF                 PNF             |
NAME      FNF                 FNF             |
CHDIR     does not apply
RMDIR     does not apply
MKDIR     does not apply

OPEN      FNF                 FNF             FNF
LOAD      |                   |               |
SAVE      |                   |               |
MERGE     |                   |               |
LIST      |                   |               |
```

EXTRA CHARACTER in filename

File names limited to 8 characters by the interpreter when an extension exists to the filename.

Example: for file which exists on the DOS level as "FEEFIFOF.BAS"

Error: kill "feefifofum.bas"

The alternative to handling this as an error is to truncate it to 8 characters:

```
Statement IBM BASIC          GW (IBM Compat) GW (GW Compat)
--------- -------            ----------      ----------
FILES     legal               legal           legal
KILL      BFName              BFName          |
NAME      BFName              |               |
CHDIR     does not apply
RMDIR     does not apply
MKDIR     does not apply

OPEN      BFNum               BFNum           |
LOAD      |                   |               |
SAVE      |                   |               |
MERGE     |                   |               |
LIST      |                   |               |
```

Known remaining incompatibilities:

The statement CHDIR "..." results in "Path not found" for IBM BASIC, whereas the other interpreters handle it as CHDIR "..".

D.5.2.2   /I Option

When IBM compatibility is selected, /I (static FDBs) will be the default. The /I option is defaulted in BASICA (there is no other option), whereas in current GW-BASIC it must be specified on the command line, if the user wishes IBM compatibility.

D.5.2.3   Multiple Line Feeds on INPUT

When compatibility is selected, multiple line feeds will *not* be mapped into a single line feed when read from a device. For example:

```
10 OPEN "FOO" FOR OUTPUT AS #1
20 PRINT #1,CHR$(&HA)+CHR$(&HA)+CHR$(&HA)
30 CLOSE #1
40 OPEN "FOO" FOR INPUT AS #1
50 INPUT #1,A$
60 PRINT A$
70 CLOSE
```

When compatibility is selected, A$ will contain three line feeds, as IBM Basic currently does. Standard GW interpreters return only one line feed.

D.5.2.4   Trailing Blanks on INPUT

When IBM compatibility is selected, the INPUT statement will NOT strip trailing blanks.

D.5.2.5   OPEN and LEN=

When IBM compatibility is selected, GW BASIC will allow an OPEN statement of the form:

```
OPEN "FOO" AS 1 LEN(10
```

Where the "(" after LEN is allowed. This is the reintroduction of a bug, whereby IBM Basic allows several characters ("(","*"," # ",")", "+", ",") in place of the normally expected equal sign. Research implies that there was a need for the "(" syntax. It is to this level that the "bug" will be allowed when IBM compatibility is selected.

D.5.2.6 Operations on Closed Files

When IBM Compatibility is selected, PRINT, WRITE or INPUT on a closed file will generate "Bad File Number" instead of its current "Bad File Mode"

D.5.2.7 OPEN LPTn:

When IBM compatibility is selected, GW BASIC will ignore all filenames included with the LPTn: device. For example:

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1
```

In this example, the "BIN" specification will be ignored. Standard GW will continue to recognize this as a binary open request. In addition:

```
OPEN "LPT1:GARBAGE" FOR OUTPUT AS #1
```

will be treated in the same manner when compatibility is selected. Standard GW will generate "Bad File Name".

D.5.2.8 Networked Printer Support

The IBM Compatible example Low Level code will be modified to support networked printer on DOS versions 3.1 and later. Required support already exists in the high level code.

D.5.3 Math and Memory

D.5.3.1 Program Segment Prefix (PSP) Data

When IBM compatibility is selected, GW-BASIC will write the Program Segment Prefix into the first 100 bytes of the Code Segment, thus mimicking the appearance of a .COM load. It will also use the first label in the data segment, BEGDSG, referenced as an offset from CS, to calculate BASIC's DS, rather than using the passed value from the EXE header.

The CS: location of the PSP has been documented to users in the Peter Norton IBM-PC book. Thus by the PSP into CS:0 and subsequent addresses, we allow users to reference the PSP through CS, as they would expect to do for a .COM like IBM BASIC.

D.5.3.2   Overflow and Division by Zero

When IBM Compatibility is selected, Overflow and Division by Zero will not be trappable errors, and the result of such calculations will be set to 1.701412E+38 (infinity?). Standard GW will continue to allow trapping, and will set the results to zero.

In addition, when error trapping is not enabled, when IBM compatibility is selected, an additional line feed will be output after the "Overflow" and "Division by Zero" error messages, to mimic IBM BASIC behavior.

D.5.4   Graphics

D.5.4.1   CLS

For all versions of GW-BASIC, CLS support will center the graphics cursor and reinitialize the DRAW turn angle (TA) unless either

    a.   The OEM routine $CLRSCN supports the user parameter CLS 2 and returns this parameter to OEM-independent code to indicate "clear text only," or

    b. $CLRSCN maps CLS <no user parameter> and ^L to parameter 2 when there is an active viewport.

Under these two circumstances no viewport or graphics tasks is handled by the OEM-independent code.

When IBM Compatibility is selected, "Syntax error" results following the execution of the CLS support code if the user has entered an integer parameter on CLS.

TA <degrees> is one of the DRAW statement commands. The change referenced is whether CLS reinitializes TA to 0 or leaves the previous user-installed TA. For example:

```
10 FOR D = 0 TO 180 STEP 4
20 DRAW "TA=D;NU50"
30 NEXT
40 CLS
50 DRAW "U100"
```

The program draws half a circle's worth of spokes. If the angle is not reset by the CLS, then the execution of line 50 draws a spoke from the center to the bottom of the screen; if the angle is reset, then the line is drawn to the

top of the screen (TA is 0, and "u" means up).

IBM and GW-BASIC do reset the turn angle upon CLS, while COMPAQ BASIC does not. The most recent GW interpreter, however, also reinitializes the viewport by reinstalling default parameters, centering the graphics cursor, and reinitializing all the DRAW parameters.

D.5.4.2 SCREEN [no parameters]

When IBM Compatibility is selected, a SCREEN statement with no parameters will generate "Missing Operand". Standard GW will continue to report "Illegal Function Call".

D.5.5 Miscellaneous

D.5.5.1 Function Key Display

The function key display line, when IBM compatibility is selected, will mimic IBM BASIC's bug in SCREEN 1, which causes the numbers and text strings to be displayed in different colors. The numbers are displayed in the "current" color, and the text is displayed in the "maximum" color.

D.5.5.2 TIME$

GW-BASIC will round up the time returned by TIME$ as IBM BASIC does. IBM Basic takes the time value returned by DOS, and rounds up the seconds if the hundredths of a second are greater than or equal to 50. GW does not, which results in visible differences in the following program:

```
10 TIME$ = "12:01:00" : PRINT TIME$
 .EI
Because of the value returned by DOS for the PRINT TIME$
statement, GW Basic currently prints "12:00:59", while IBM
returns "12:01:00".
```

D.5.5.3 CHAIN to Non-existent Line Number

When IBM compatibility is selected, CHAINing to a non- existent line number will return "Illegal Function Call". When compatibility is not selected, it will return its current "Undefined Line Number".

D.5.5.4 CHAIN and OPTION BASE

The actions of OPTION BASE across CHAINs are defined as follows.

For all versions:

Within a program, repeated OPTION BASE statements are allowed if they are the same value, else "Duplicate Definition" results. This is true regardless of whether an array has been dimensioned.

1. If there is an array defined in the chaining program, then no OPTION BASE statement which changes the OPTION BASE is permitted in a chained-to program when variables are passed to that program. This is whether or not there was an explicit OPTION BASE statement in the chaining program. If there was no explicit statement, then the option base defaults to 0. Such a statement in the chained-to program will result in "Duplicate Definition". OPTION BASE statements in the chained-to program which do not change the option base are allowed.

When GW version-to-version compatibility is selected:

- If there is an OPTION BASE statement in the chaining program, but no array is passed to the chained-to program, then the option base defaults to 0 in the chained-to program; however, an OPTION BASE statement is permitted and may reassign the option base to 1 if it precedes an array definition in the chained-to program.

When IBM compatibility is selected:

- The option base, whether implicit (0) or explicit (via an OPTION BASE statement), will be preserved across CHAIN or CHAIN MERGE. However, if there is no array passed to the chained-to program, then that chained-to program may redefine the OPTION BASE via an OPTION BASE statement before it defines its own arrays.

D.5.5.5 BSAVE & Protected Files

When IBM compatibility is selected, the interpreter will *not* prevent
BSAVE or **BLOAD** in direct mode when the current program is pro-
tected. When IBM compatibility is not selected, "Illegal Function Call" will
result.

Index